A UNIFIED APPROACH FOR FAULT-TOLERANCE IN COMMUNICATION PROTOCOLS

Anjali Agarwal

A Thesis

in

the Department of

Electrical and Computer Engineering

in Partial Fulfilment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montréal, Québec, Canada

December 1995

ISBN 0-612-10811-2

ABSTRACT

A Unified Approach for Fault-Tolerance in Communication Protocols

Anjali Agarwal, Ph.D.

Concordia University, 1995

The goal of this research is to provide a unified approach to fault-tolerance in communications systems under a model of transient failures by formally incorporating the states and transitions for fault-tolerance into the specification and design phases of the communication software development life cycle. Historically, researchers have tended to address the wide variety of phenomena within fault tolerance in the area of distributed database and distributed computing by countering the effects of their individual causes. Not much work has been done in the field of providing fault-tolerance to communication protocols, especially at the specification level. Since protocols include a large amount of abnormal processing triggered by transient failures, high reliability and performance in the presence of such events are required for such protocols. Therefore, after a failure in the processor running the software or the process itself or a crash of the local memory of the process, which may lead to an unstable and illegal system state, the protocol must be able to recover and continue its execution starting from a legal state. A protocol that possesses such a feature is called a fault-tolerant protocol. In the context of such fault-tolerant protocols it is assumed that the starting point in the protocol development life cycle is a complete specification and an error-free design.

An understanding of fault-tolerance based on checkpointing and rollback recovery in distributed system environments is introduced and the related work is reviewed, and a better approach to checkpointing and rollback recovery is proposed. The idea of checkpointing and rollback recovery in distributed systems is applied to obtain a scheme for recoverability of protocols. The protocols after recovery can return not only to an initial state, but also to an intermediate state that was reached in the past, while retaining consistency in the exchange of messages. The rollback recovery procedures are then incorporated into the specification and design phases of the communication protocols. Their validated protocol specifications are provided.

ACKNOWLEDGEMENTS

It is my great pleasure to express sincere gratitude to others who have contributed immensely to the successful completion of this work.

I have a deep feeling of indebtedness to my supervisor, Dr. J.W. Atwood, for his constant confidence, support, encouragement, guidance and for providing me financial support. His suggestions and criticisms from time to time are appreciated, and more importantly, his invaluable friendship will never be forgotten.

I am also thankful to my ex-supervisor, Dr. Kassem Saleh, for introducing the thesis topic to me and for providing me financial support during the first year of this thesis preparation. I would also like to thank Dr. K. Thulasiraman for taking charge of being my supervisor during the difficult time of finding a new supervisor when Dr. Saleh resigned, and for advising me to consider Dr. Atwood as my supervisor later. I would also like to thank him for providing me financial support during the second year of my research.

Many thanks are due to all my friends for helping to establish a convivial environment in which to work and for providing the needed moral support during the difficult stages of this work.

Finally, I thank Ambrish, my husband, to whom I owe much for his unending patience, encouragement and understanding without which I could hardly have completed this work. Special thanks are to my two lovely daughters, Niharika and Radhika, for understanding me for not giving them full attention during their growing years. I also missed a few of Radhika's first things in life, since we had to leave her with her grandmother when we moved to Canada for this work.

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

ACK	acknowledgement message
CFSM	communicating finite state machine
CHK	checkpoint message
CHKF	checkpoint upon failure message
CP	checkpoint
d	depth of minimum spanning tree
EI	event index
EREI	error event index
EXECEV	executed event
FIFO	first-in first-out
GCP	global checkpoint
ICP	independent checkpoint
IN#	incarnation number
IR	initiate recovery message
L	maximally reachable state
M	number of processes failed
MREI	maximally reachable event index
m,n	application message
N	number of total processes in the system
P	process
PFR	permanent failure recovery
rcm	recovery control message
R	maximally executable sequence
RM	recovery message
RMack	acknowledgement message to recovery message

RP	recovery point
Si	local states of a process (i>=0)
t,T	time
umsg	unspecified message

CHAPTER 1

INTRODUCTION

"Error does not become truth by reason of multiplied propagation, nor does truth become error because nobody will see it."

Mahatma Gandhi

1.1. Motivation

The design of fault-tolerant hardware is a mature area of computer engineering and architecture. Several techniques exist to ensure that a hardware failure does not affect the normal operation in a computer system. The problem of designing fault-tolerant software has been studied more recently. In particular, the design of distributed systems and real time software is a more complex and critical problem.

The goal of this research is to provide a unified approach to fault-tolerance in communications systems under a model of transient failures by formally incorporating them into the specification and design phases of the communication software development life cycle. Additional states and transitions should be incorporated into the specification phase itself to provide fault tolerance to a system of communicating entities, to bring the system into a consistent state should a transient failure occur.

Historically, researchers have tended to address the wide variety of phenomena within fault tolerance in the area of distributed database and distributed computing by countering the effects of their individual causes. Not much work has been done in the field of providing fault-tolerance to communication protocols, especially at the specification level. Since protocols include a large amount of abnormal processing triggered by transient failures, high reliability and performance in the presence of such events are required for such protocols.

The design of self-stabilizing communication protocols has been studied by Gouda [GoMu91], where self-stabilization has been taken as an approach to fault-tolerance. A communication protocol is defined as a set of syntactic and semantic rules that are used to govern the communication between two or more communicating entities. In a distributed computer environment communication protocols play an important role. Because of the complexity and criticality of communication software, it is necessary to obtain designs for protocols that are free from general types of errors such as deadlocks and unspecified receptions. In addition, after a failure in the processor running the software or the process itself or a crash of the local memory of the process, which may lead to an unstable and illegal system state, the protocol must be able to recover and continue its execution starting from a legal state. A protocol that possess such a feature is called a fault-tolerant protocol. In the context of such fault-tolerant protocols it is assumed that the starting point in the protocol development life cycle is a complete specification and an error-free design.

If we are dealing with self-stabilizing communication protocols, then we should be willing to tolerate the temporary violation of a system specification. The protocols should be non-terminating, since termination is one of the factors that can prevent self-stabilization, and there should be an infinite number of safe states [Schn93]. The solution to self-stabilization is based on invariants that must be verified in the design. The problem is that such invariants are protocol or problem specific. They are flexible but are difficult to use. Expertise in protocols is required to correct an erroneous invariant, to encompass all failure modes within the invariant, and to establish time bounds for recovery. However, there should be no such requirement for providing fault-tolerance in communication protocols. If we can deal with the processes being initialized to a consistent state, the processes can be brought to a legal state should a transient failure occur, by applying checkpointing and backward error recovery procedures.

1.2. Overview

The research includes several contributions :

1) First the area of fault-tolerance based on checkpointing and rollback recovery in distributed system environments is introduced and related work is reviewed. To start with, the concepts and requirements for our checkpointing and rollback recovery algorithms are defined, and an algorithm is proposed to obtain the maximally reachable event index, MREI(i), which represents the event indices corresponding to sending/receiving messages that other processes must have reached before process P_i in a system of distributed processes executes its event.

2) Second the problem of fault-tolerance in real-time distributed systems is addressed by providing efficient algorithmic procedures for checkpointing and rollback recovery in such systems. Even when the communication network is reliable and maintains the order of messages, any kind of processor errors, which may not be detected immediately, could contaminate the system resulting in failure of process(es). Two approaches to fault-tolerance based on recovery are then considered.

(i) In the first approach, efficient and fault-tolerant algorithmic procedures for obtaining coordinated global checkpoints and rollback recovery are proposed. Each process is required to record the contextual information exchanged between the distributed system components during the normal system progress, and the current local state on stable storage.

- The checkpointing algorithms can be initiated by any process in the system, or upon failure of one or more component processes as part of a backward recovery procedure. Our algorithms return the most recent and consistent checkpoint, and require stable storage to save the current MREI and the local state for every send/receive event. Stable storage is also required to save the message received by the receiver process. The algorithms do not interfere with the progress of the distributed system application. Furthermore, the Domino effect is avoided since obtaining a consistent checkpoint is always guaranteed.

- Utilizing these global checkpoints, we also present optimal backward recovery procedures, which require minimal rollback after failures. Livelock problems associated with rollback recovery are avoided since the consistency of message exchanges is retained when a rollback to a consistent checkpoint is executed.

To illustrate the checkpointing and recovery procedures, a time diagram with the exchange of messages is considered. Different scenarios are considered for a process failing/detecting an error during different stages of checkpointing.

(ii) In the second approach, our recovery procedures do not require the application of checkpointing procedures to obtain coordinated global checkpoints, but require each process to record the contextual information exchanged between the distributed system components during the normal system progress,

- in the first case, on stable storage whose requirement is bounded and minimal and

- in the second case, on volatile storage, further minimizing the requirement for stable storage and the time needed to access the contextual information from stable storage. Each process also saves its local and independent checkpoints in stable storage.

The recovery procedures provide a global consistent state before the occurence of error(s) in failed process(es) and before the effects of the errors reach other processes. In the global consistent state, the recovery point computed locally by failed process(es) is mutually consistent with the recovery points computed by other non-failed processes. Also, only those processes are required to restart that are affected by error propagation, and the time needed to revert to normal operation is found to be minimal, thus enhancing the real-time responsiveness of such systems. Our recovery procedure can be used to recover from any number of process failures in the system, including a total failure of all processes. Proofs of correctness of our procedure are provided; in particular, the absence of orphan messages, message losses and duplications is shown. Also, pre-rollback messages are taken care of.

An interaction sequence diagram and its time sequence diagram showing its event index tuples is considered for a distributed system example consisting of four communicating finite state machines. The recovery control messages exchanged after failure detection are also shown.

3) Next the idea of rollback recovery in distributed systems is applied to address the problem of designing stabilizing computer communication protocols modeled by communicating finite state machines. A communication protocol is said to be stabilizing, if starting from or reaching any illegal state, the protocol will eventually reach a legal (or consistent) state, and resume its normal execution. To achieve stabilization, the protocol must be able to detect the error, and then it must recover from that error and revert back to a legal protocol state. The later issue related to recovery is tackled here and efficient procedures for the recovery in communications protocols are described. The recovery procedures do not require periodic global checkpointing procedures and therefore, are less intrusive. They require less time for rollback and fewer recovery control messages than other procedures. Only a minimum number of processes will roll back, and a minimal number of protocol messages will be retransmitted during recovery. Our procedure, in the first case, requires the stable

storage to be used to record contextual information exchanged during the progress of the protocol. A protocol example consisting of six communicating finite state machines specifying a part of the ISDN user part of Common Signalling System (CSS7) is considered and recovery control messages required are shown using the time sequence diagram. In the second case, the volatile storage is used to log the application messages and the stable storage to save the independent checkpoints of the processes. Our recovery procedure can be used to recover from any number of transient errors in the system. Our procedure is compared with the existing approaches for handling the errors. A protocol example of four interacting processes is considered to illustrate the recovery procedures by transition diagrams when operational errors change the state of the system by corrupting the local state of a process(es) as represented by memory or program counter.

Our recovery procedure has also been modeled in PROMELA, a language to describe validation models, which shows the syntactic correctness of our recovery protocol design.

The recovery procedures can be incorporated into the specification and design phases of communication protocols. All the design methods for fault-tolerant protocols described are application-independent. Since they require only the protocol structure and exchange of local information, they are regarded as unified design methods for any protocol and all failure modes. However, the procedures and memory required are assumed to be fault-tolerant.

1.3. Organisation

This thesis is organised according to our contributions. Chapter 2 deals with the basic concepts and definitions for fault-tolerance based on checkpointing and rollback-recovery, and its related work is reviewed. The distributed system considered in the thesis is introduced, and its space time model and system interaction diagram shown. The concepts on which our checkpointing and rollback-recovery algorithms are based are then described, and the algorithm to obtain the maximally reachable event index is presented.

In Chapter 3 we describe our proposed procedures for fault-tolerance in distributed systems based on checkpointing and recovery. Two approaches are considered. In the first approach, the processes communicate to obtain coordinated global checkpoints upon failure or otherwise, in addition to logging the messages received and saving the local states on stable storage. The system rolls back to these global checkpoints in case of failures maintaining the consistency of exchanged messages. In the second approach we present recovery procedures that do not require coordinated checkpointing procedures but require only the messages to be logged on stable/volatile storage with the contextual information exchanged during the normal system progress.

Chapter 4 is about fault-tolerance in communication protocols. We first introduce the formal model for communication protocols based on communicating finite state machines. We then discuss the issues related to fault-tolerant protocol design and describe the failure model that we will use in our discussion on fault-tolerance. Finally, for the failure model considered, we propose recovery procedures based on the second approach of Chapter 3.

The design of fault-tolerant protocols is presented in Chapter 5. Different methods using conventional, verification and synthesis approach for the design of fault-tolerant protocols are surveyed. Our research is based on incorporating the checkpointing and rollback-recovery procedures proposed in Chapter 4 to provide fault-tolerant protocols using the synthesis method. An example of such a fault-tolerant protocol obtained is presented by displaying the transition diagram for its recovery procedure. Validation of this transition diagram using PROMELA is also provided.

Finally in Chapter 6, some conclusions are made on the work done and some lines of future research considered. The next section gives a brief overview of fault-tolerant computing before going into more detail in the subsequent chapters.

1.4. Fault-Tolerant Computing

A fault-tolerant computing system is one that can provide its specified services in the presence of a predetermined number and class of failures. The basic idea behind building in a fault-tolerance capability is to provide the system with extra (redundant) resources, beyond the minimum needed to achieve the computing requirements. These extra resources can help to overcome the effects of a malfunction. The redundancy can take the form of extra hardware, which can vote out an erroneous signal or switch in a spare to replace a failing subsystem, or additional software, which can allow successful reexecution of a program following detection of a failure caused by unknown factors.

The terms failure, fault and error have different meanings when applied to computer systems:

- a **failure** denotes an element's inability to perform its designed function because of errors in the element or its environment, which in turn are caused by various faults. It is a violation of specifications and assumptions. An invalid transition within the component leads to component failure, whereas if all the components meet their specifications when an invalid transition takes place, the failure was a design failure.

- a **fault** is an anomalous physical condition. It is caused due to design errors such as mistakes in system specification or design and implementation; external disturbances such as harsh environmental conditions, unanticipated inputs or system misuse.

- an **error** is a manifestation of a fault in a system, in which the logical state of an element differs from its intended value. An error occurs only when a fault is "sensitized", that is, for a particular system state and input excitation, an incorrect next state and/or output results.

The following summarizes the relationship between the three,

$$\text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure}$$

A fault may be localized in a given component or distributed to more than one component. In a system in which the components interact with each other, erroneous information produced by one faulty component can be propogated to other components. This type of propogation is called error propogation.

In general, there are four phases in which fault-tolerant techniques can be implemented. These are error detection, damage confinement and assessment, error recovery, and fault treatment [LeAn90, RLT78].

Error Detection - Since it is difficult to detect faults directly, a detection process usually tries to detect errors caused by faults. Error detection techniques are used for identifying erroneous states of a system. Unfortunately, an arbitrary period of time may pass before fault detection occurs. This time is called fault latency.

Damage Assessment - Because of error propogation, and the consequent damages that can be spread over the system, some techniques are necessary for assessing how much the system state has been damaged for an appropriate recovery. These techniques are called damage assessment techniques.

Error Recovery - Error recovery techniques attempt to move a system from an error state to an error-free one. There are two approaches for the error recovery techniques: backward error recovery and forward error recovery [LeAn90, Mili90]. Techniques based on forward error recovery try to use the current error state to construct a new state in the hope that it is error-free, so tnat the normal computation can be resumed. Techniques based on backward error recovery try to recover a system from errors by discarding the current system state, restoring the system to a prior known consistent (error-free) state, and restarting execution. Therefore, they require state information to be saved during normal computation. Since a backward error recovery restores a system to a valid prior system state, a recovery is possible from errors of unknown origin and propogation characteristics.

Fault Treatment - Once error recovery has been done, it is necessary to enable a system to provide its specified services. If faults are transient in duration, no special treatment may be required.

The thesis is mainly concerned about providing backward error recovery procedures considering the effects of error propagation and the damage caused by it.

CHAPTER 2

FAULT-TOLERANCE BASED ON CHECKPOINTING AND ROLLBACK RECOVERY

2.1. Introduction

The ability to restart the execution of a process is very important for applications that depend on the progress of the system in the presence of failures. Re-execution from the start of a process is generally straightforward. However, it is usually beneficial to restart the execution from a pre-defined point in the process, rather than from the beginning. This pre-defined point is called a *checkpoint*.

Checkpointing in a distributed system amounts to obtaining a consistent global state or a consistent set of checkpoints during the progress of the distributed system computations. This is an inherently difficult task because of the distributed nature of the progressing system computations and the arbitrary message-transfer delays usually associated with geographic distribution. A state of the distributed system (also called global state, system state, or distributed snapshot) consists of the current state of each of the component processes and the contents of the channels linking these processes. During execution, the identification of the system state is useful for establishing checkpoints for recovery and restart [Raynall88, Morg85] and for detecting abnormal conditions such as system deadlocks and termination [ChLa85]. A consistent and legal system state can be used as a recovery point in the case of a future system failure. Intuitively, when a failure occurs, the processes roll back to their most recent consistent checkpoint by restarting from the state saved in that checkpoint and resuming execution from it.

2.2. Distributed System Model

A distributed system consists of a finite number of loosely coupled processes running on a finite number of processors that exchange messages over communication links. These processes form the nodes of a strongly connected network. To recover from process errors and failures, and restore the system to a consistent state, we use two types of logs—a volatile log and a stable log Accessing volatile logs requires less time, but the contents of a volatile log are lost if the corresponding processor is in an error state leading to its failure. The volatile/stable log is used to save critical contextual information required for our checkpointing and recovery procedures. We assume that the underlying computation or the application program is event-driven, with the processes communicating with each other by hanging messages through a communication subsystem modeled by unidirectional first-in-first-out (FIFO) channels of unbounded capacities. Since messages are delivered through channels, each delivery takes a finite amount of time but has an arbitrary delay. The channels are assumed to be reliable, meaning that they do not duplicate, eliminate, or corrupt messages. Processes are considered to be deterministic, meaning that replaying a sequence of events from a state will consistently reach the same final state. We also assume that the processes are non-fail-stop, meaning that an error occuring in a process may not be detected immediately by the recovery system. Therefore, an error that occurs in one process can contaminate further checkpoints and states of itself and other processes in the system. We also assume that once an error is detected by a process, the process will be able to determine the actual point of failure, and will be able to initiate the recovery procedure.

Due to the autonomous behaviour of processes and arbitrary communication delays, any single process in a distributed system cannot capture the complete system state instantaneously. Therefore, gathering information on process states in different processors and channel states, that is, the global state, may require solving many problems existing in distributed systems.

2.2.1. Space Time Model of Distributed Computations

In the space time model (STM), a distributed system is viewed as a collection of communicating sequential processes, each consisting of a sequence of events. An event could represent an execution of a function or a single instruction, or the sending and receiving of messages. We assume that the ordering of events is strictly governed by partial ordering on a set of events [Lam78].

Figure 2.1 shows a space time diagram of a distributed system example. A process is represented by a horizontal line called a process line. Nodes in each process line represent events. For any two events a and b, an event a precedes or "happens before" an event b if and only if $a < b$. The precedence relation $<$ is defined as follows:

1. if a and b are events in the same process then $a < b$ implies a happens before b.
2. if a is a send event in one process and b is its corresponding receive event in another process, then $a < b$ by their causality relationship.
3. if a, b, c are events and $a < b$ and $b < c$, then $a < c$ by transitivity.
4. two distinct events a and b are concurrent if neither $a < b$ nor $b < a$.

2.2.2. System Interaction Diagram

A system interaction diagram shows the logical interrelationships among the processes taking part in the distributed system computations [SAAA94]. This diagram is a directed graph consisting of a set of vertices and a set of directed edges. Each vertex corresponds to a process in the system. A directed edge between two

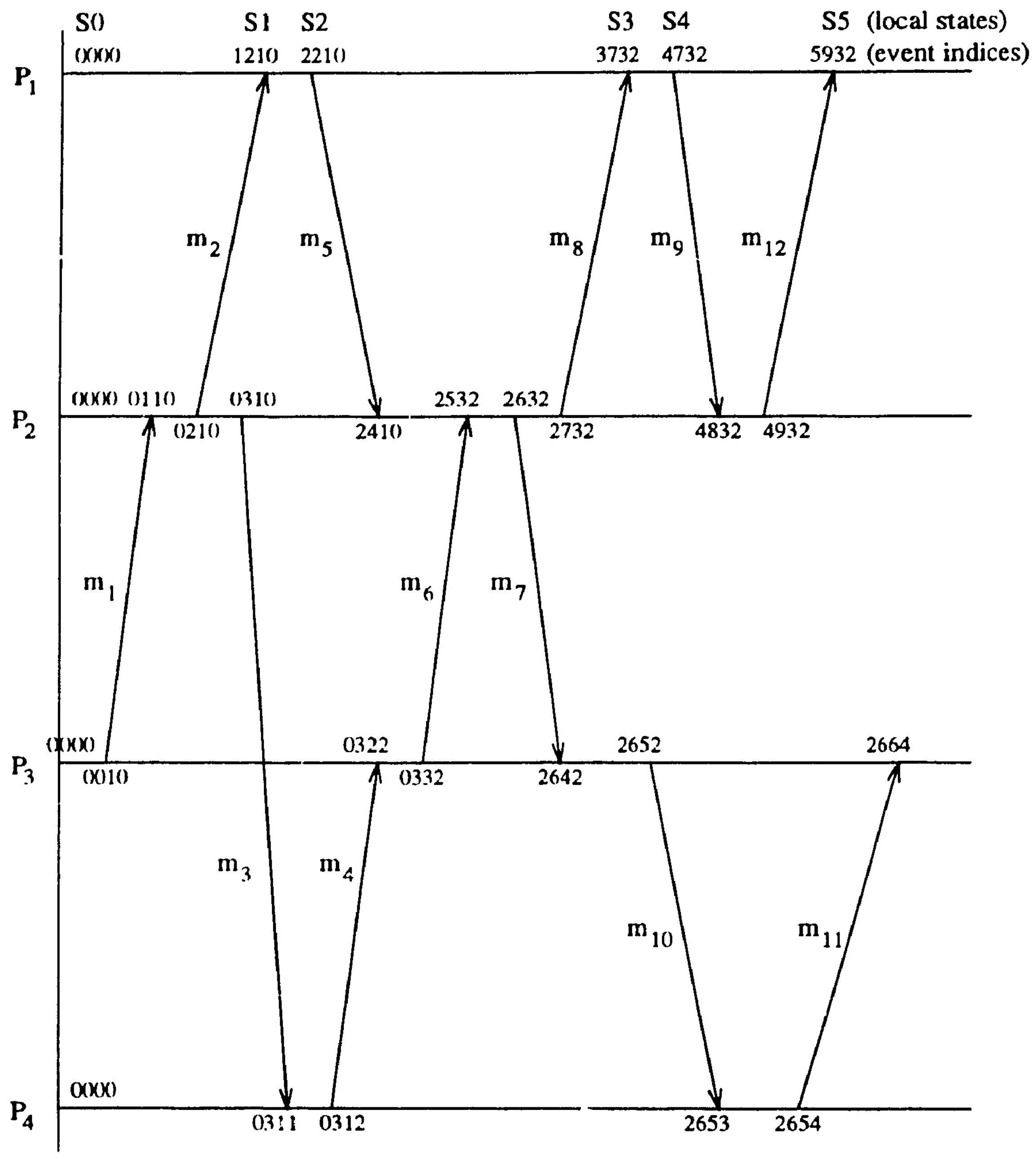


Figure 2.1 : Space Time Diagram of a Distributed System example

processes, say from P_j to P_k, exists if there exists a message flow from process P_j to process P_k. For example, if we consider the distributed system specification shown in Figure 2.1, we need to have the processes connected in a strongly connected network with the system interaction diagram as shown in Figure 2.2.

We also assume that if a message is to be sent from P_j to P_k and there does not exist a channel between P_j and P_k, the message is sent through the shortest possible path between P_j and P_k in the strongly connected network. If it takes one time unit for the message to reach between two nodes, then the time required for the message from P_j to reach P_k is a function of the shortest path. If a process is required to communicate with all other processes, it will take longer if the underlying communication system is not fully connected.

2.3. Consistent Checkpointing and Rollback Recovery

Definition 1: A **checkpoint** is the saved state of a single process stored in a form such that the process can restart its execution from the point in time when the checkpoint was created.

Definition 2: **Checkpointing** is the process of saving process states into checkpoints.

Checkpoints contain all information needed to restart the process in which the checkpoint was created. When the process is restarted, its current state is discarded and the state saved in the checkpoint is restored.

Definition 3: The restoring of the state of one or more processes to the state previously stored in a checkpoint is called **Rollback**.

A set of checkpoints, one per process in the distributed system, is said to be inconsistent if the saved states form an inconsistent system state. A system state is

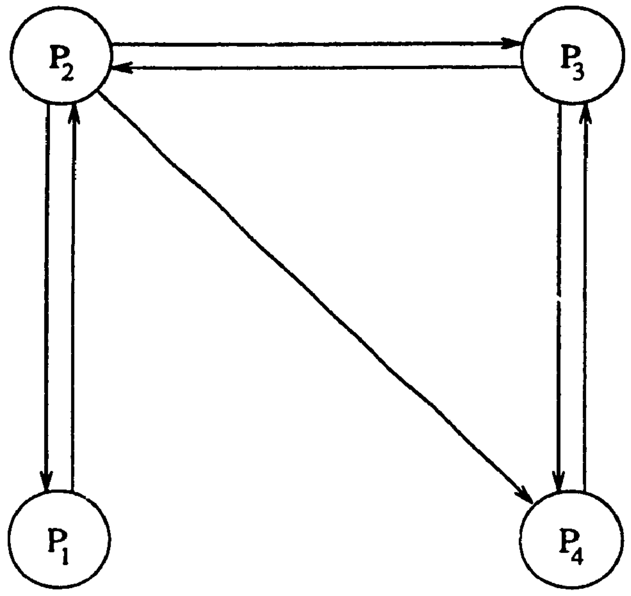


Figure 2.2 : System Interaction Diagram for Figure 2.1

inconsistent if a process in the system has received a message that has not been sent yet in this system state. However, the system state is regarded as consistent if a message is already sent by one process but not yet received by the other process. Therefor e, when a process wishes to checkpoint its state, it must be sure that when it decides to roll back to this checkpoint at a later time, it will be in a consistent state with respect to the other processes in the system. Consistency upon rollback is ensured by guaranteeing that any dependencies on messages are reflected in the checkpoint along with the actual state of the process.

The rollback of a single process may affect the execution of other processes in the distributed system. The rollback to a checkpoint will "undo" any communication that may have occurred since the checkpoint was created. Consider the execution of two interacting processes as shown in Figure 2.3. Suppose that the process P_2 creates a checkpoint at time t_1 and afterwards receives a message m from P_1. At time t_2, P_2 performs a rollback to the previously saved checkpoint and repeats the execution starting from that state. The system state is represented by the line joining (P_1, t_2) and (P_2, t_1), and it meets the definition of consistency. However, if P_2 is rolled back to point t_1 and then re-executed, it will be waiting for message m, which will not arrive if P_1 does not also roll back its state to a point in its execution prior to the sending of m. The system recovers from a consistent state, but message m is lost.

Another serious consequence of interaction on checkpointing is depicted in Figure 2.4. In this figure, process P_1 takes a checkpoint at time t_1 and then sends a message to process P_2. After receiving this message, P_2 takes a checkpoint at time t_2. Subsequently, P_1 fails and restarts from the checkpoint taken at t_1. The system state at P_1's restart is inconsistent, because P_1's local state shows that no message has been sent to P_2, while P_2's local state shows that a message from P_1 has been received. This problem is also known as the orphan message problem.

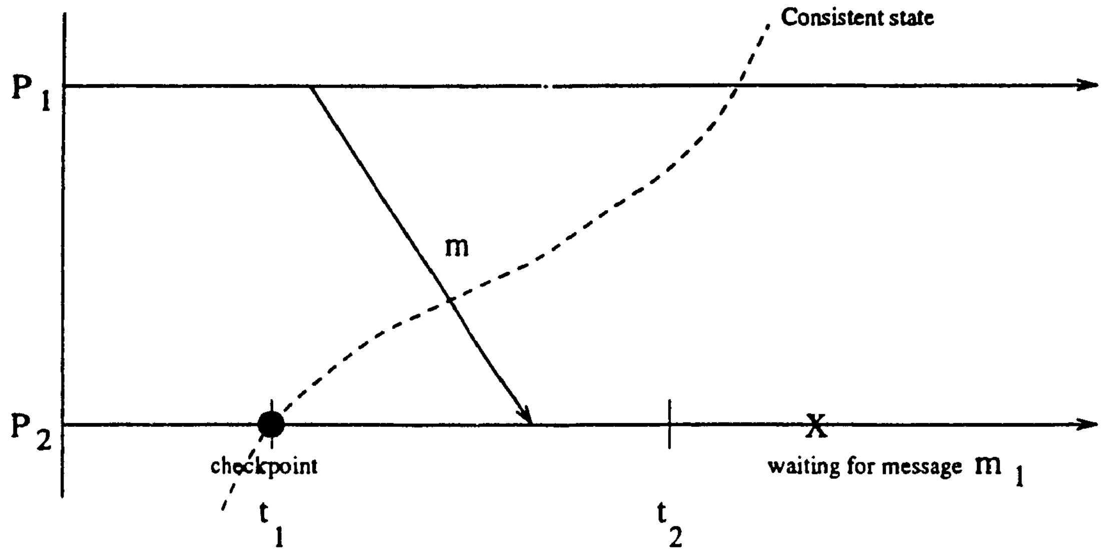


Figure 2.3 : Inconsistent Rollback of Interacting Processes

(Message Loss Problem)

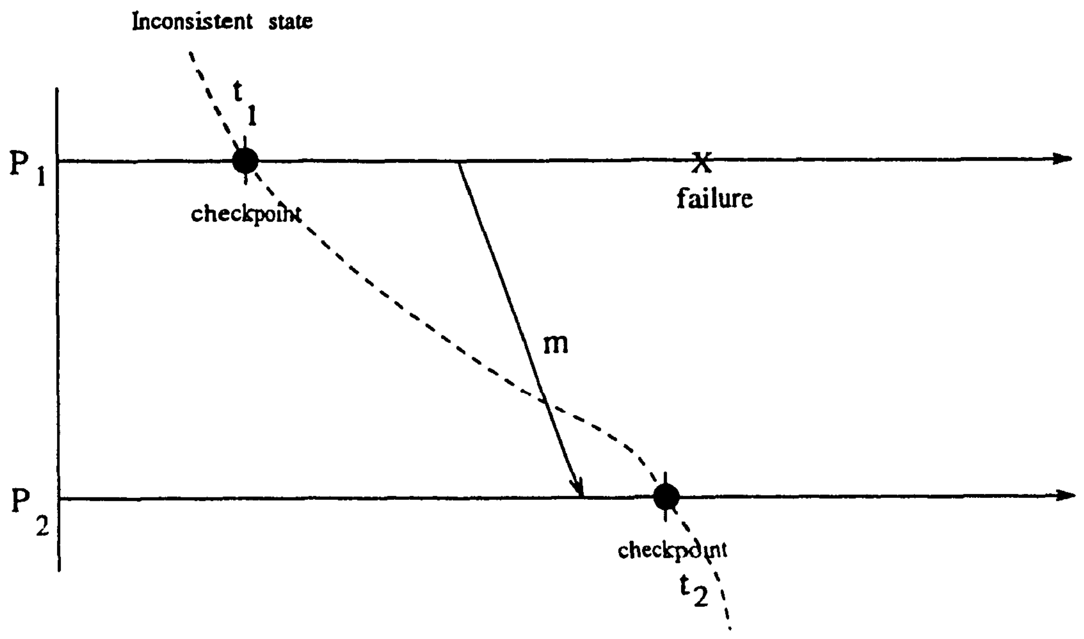


Figure 2.4 : Inconsistent State due to Orphan Message Problem

As the system executes, the message loss problem can be avoided by recording the messages on stable storage in a message log. Each message carries with it an event index for the sending process at the time the message was sent. A message is called logged if and only if its data and the event indices for the sending and the receiving processes are recorded on storage. Logged messages remain on storage until they are no longer needed for recovery from any possible future failure of the system. Recovery from an inconsistent state as shown in Figure 2.4 is possible if the orphan messages can be distinguished and are not sent by P_1 after recovery since these messages are already on the message log for P_2. Processes must be rolled back in such a way as to insure that any two processes are in a consistent state with respect to each other with no message lost, i.e., they agree on which messages have been sent and which ones have not.

Definition 4: Any set of checkpoints (no more than one checkpoint per process) which taken collectively form a consistent state of a distributed system is called a **Recovery Line**.

2.3.1. Requirements for Efficient Checkpointing and Recovery Procedures

Checkpointing and rollback recovery procedures can be used to provide a method of restarting an application from a point in time prior to the occurrence of a fault thereby rendering the program more resilient to faults. It has been used in applications such as distributed programs and distributed database systems. But these procedures must follow certain requirements [FrTa89, SaAg93a].

An ideal checkpointing procedure should:

1. be nonintrusive, meaning that when the procedure is executed, there should be no interference with the underlying distributed system computations, processes should always be able to exchange system application messages and there is no

visible effect on the performance of the system;

2. obtain an efficient checkpoint, meaning that the obtained system state should be as close as possible to the system state when the procedure stops;
3. obtain a meaningful state, meaning that the obtained system state should be a state that has actually occurred during previous computations of the system;
4. incur low overhead minimizing the time lost during checkpointing and disk space for storing checkpoints;
5. be fault-tolerant, meaning that if during the execution of the procedure a failure occurs in the system, the procedures should still terminate and return a consistent system state or checkpoint.

An ideal recovery procedure should:

1. provide us with a maximum global consistent state, i.e., for the failed process, the state before the occurrence of error, and, for the other processes, the state before the effects of the error had propagated to that process;
2. require a minimal number of processes to rollback, meaning that in addition to the process where error occurs, only the processes affected by the error need to roll back to their respective recovery points;
3. use a minimal number of recovery related messages;
4. require a minimal number of application messages to be retransmitted;
5. use a minimal amount of storage for its implementation; and
6. recover from any number of process failures in the system.

2.3.2. Problems Related to Checkpointing and Rollback Recovery

This Section outlines some problems that make checkpointing and recovering the state of a system very difficult. They are as follows [Sarr93, KoTo87, SaAg93b]:

1) **Lack of global time**: Distributed systems suffer from the problem of having no global time scale. Events occurring in a traditional sequential program will always be totally ordered by physical time. Unfortunately, this is not the case in a distributed system. Therefore, some method of artificially ordering events in different processes is needed in order to discuss the creation of meaningful checkpoints. In checkpointing algorithms a partial order is imposed on all of the events using the "happens before" relation as described in Section 2.2.1.

2) **Communication delay**: If the state of a single-process program is checkpointed, it can be done so instantaneously since there is no need to coordinate its checkpoint creation with any other processes as there is no outside interaction. However, in multiple-process applications the fact that there is no global time scale, implies that it is impossible for system wide states to be saved instantaneously. Therefore, some communication protocol must be employed in order to save states that are consistent across several processes. However, in distributed systems, there is an inherent delay between the time that a message is sent and the time at which it is received at some remote process. Since checkpoints cannot be created according to a global clock, and messages incur a delay, it is impossible to save and restore states of several processes instantaneously.

3) **Frequency of checkpointing**: This really depends on the application. The two things to consider when deciding on the checkpoint frequency are the need to minimize the amount of computation to be rolled back, and the overhead of the actual checkpointing operation. If checkpoints are taken often, the system performance will degrade but recovery time will be decreased. On the other hand, if checkpoints are taken less often, system performance will be affected to a lesser extent but a penalty will be incurred at the time of rollback and recovery. The designers of checkpoint and rollback recovery must weigh the advantages and disadvantages of checkpoint

frequency on the basis of how likely it is that the system will have to be rolled back. This can be considered as an optimization problem as done in [ChKr88].

4) **Pre-rollback messages**: The communication delay inherent in distributed systems introduces another obstacle to rollback and recovery. When a process has rolled back its state, some messages that it has sent prior to its rollback may still be in transit (i.e., the messages may not have been received at their destination yet). These messages would no longer be valid since they were sent before the sender had changed its state.

Definition 5: Messages which are sent by a process prior to its rollback but are received by another process after the rollback is complete are called **pre-rollback** messages.

It is important for checkpoint and rollback recovery algorithms to make sure that these pre-rollback messages are somehow flushed out of the system so that they are not mistaken for valid message resends upon rollback. Consider the system execution in Figure 2.5. After the recovery line is created, messages are sent between processes P_1 and P_2. After the rollback, P_1 resends message m_1 as m_1' and P_2 resends message m_2 as m_2'. However, message m_3, which was sent from P_2, was in transit while the rollback was taking place and arrives at P_1 after the rollback. If m_3 is not detected as a pre-rollback message, P_1 accepts message m_3 thinking that it is actually receiving m_2 and the whole system execution is now in error.

5) **The domino effect**: Restarting a system from a set of inconsistent checkpoints may cause the biggest problems in checkpointing and rollback recovery, the domino effect problem [Ran75] as illustrated in Figure 2.6. The domino effect refers to the avalanche rollback of processes to their initial state due to the rollback of one process. In Figure 2.6, processes P_1 and P_2 have independently taken a sequence of checkpoints. The interleaving of messages and checkpoints leaves no consistent set of checkpoints for P_1 and P_2, except the initial one at (X_0, Y_0). Consequently, after

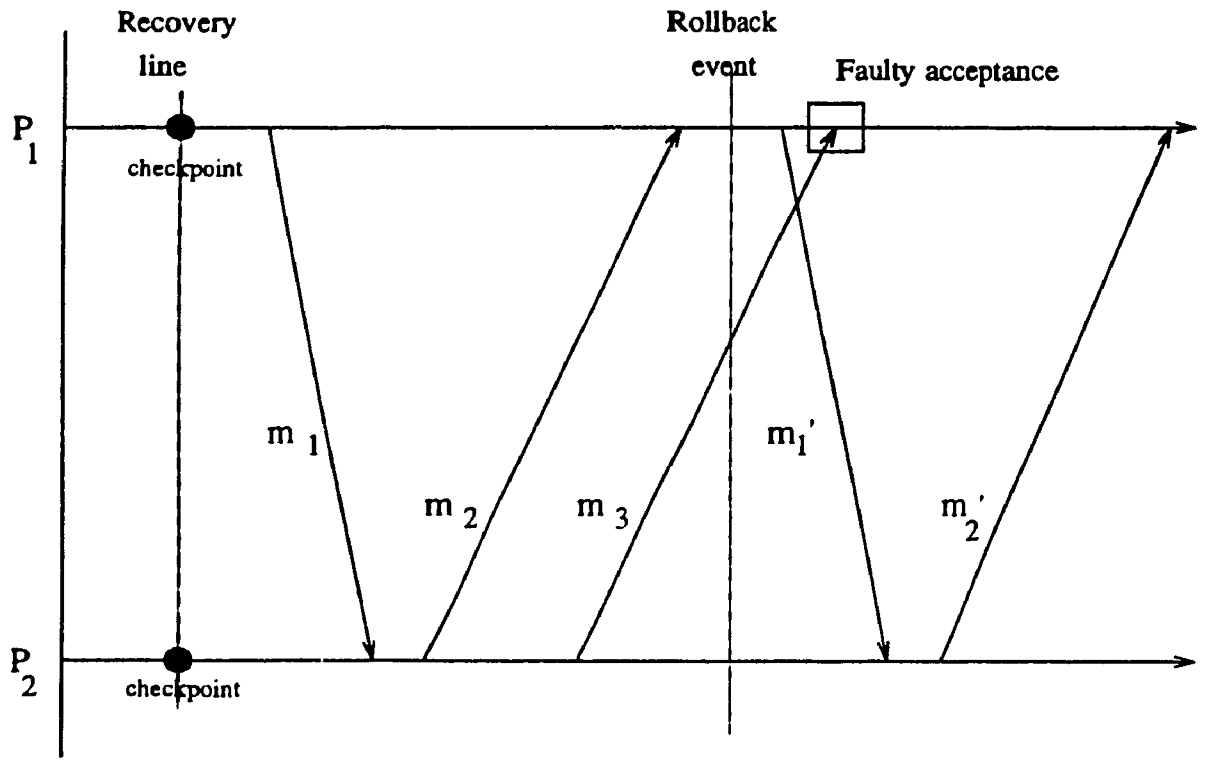


Figure 2.5 : Faulty-acceptance of a Pre-Rollback Message

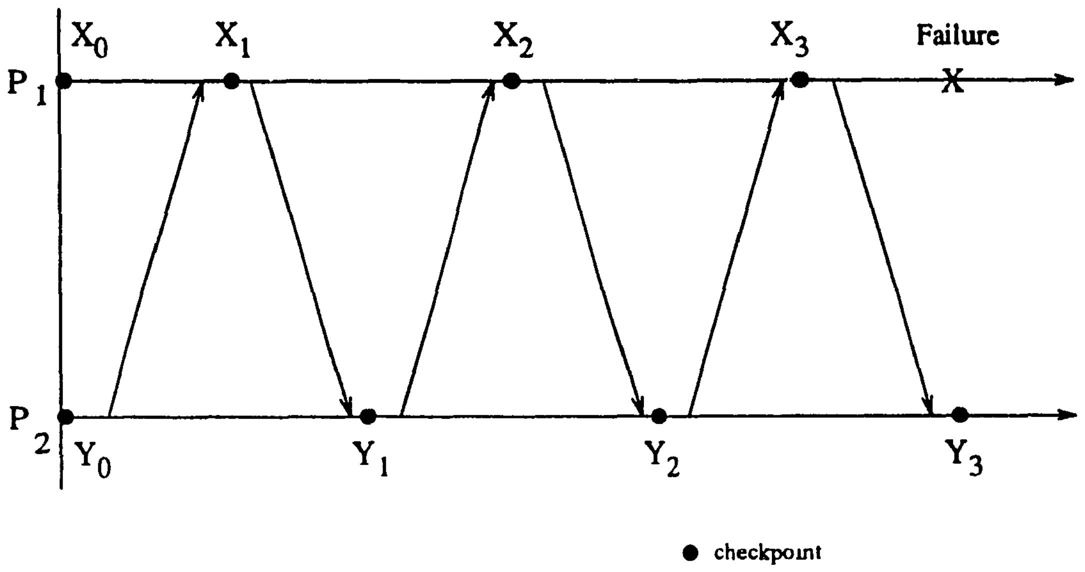


Figure 2.6 : "Domino-effect" following a Failure

P_1 fails, both P_1 and P_2 must roll back to the beginning of the computation. For time-critical applications that require a guaranteed rate of progress, such as real time process control, this behaviour results in unacceptable delays. Message logging has been proposed to avoid such a problem.

6) **Livelock**: When a process rolls back to its checkpoint, it notifies all other processes to also roll back to their respective checkpoints. Due to communication delay all processes cannot recover simultaneously. Recovering processes asynchronously can introduce livelocks; i.e., situations in which a single failure can cause an infinite number of rollbacks, preventing the system from making progress. Such a situation is illustrated in Figure 2.7. Process P_1 fails before receiving the message n_1, rolls back to its checkpoint W, and notifies P_2. Then P_1 recovers, sends m_2 and receives n_1. P_1 now has no record of sending m_1, whereas P_2 has a record of its receipt. The global state is therefore inconsistent. P_2 must also roll back to its checkpoint Y to forget the receipt of m_1. Now P_2 has no record of sending n_1 whereas P_1 has a record of receiving n_1. Hence, P_1must roll back a second time to restore consistency. Furthermore, P_2 sends n_2 and receives m_2, after it recovers. Message n_2 is received by P_1 after it rolls back. However, as a result of this second rollback, P_1 forgets the sending of m_2. Therefore P_2 must roll back a second time to restore consistency. And this second rollback of P_1 will cause the third rollback of P_1. P_1 and P_2 are forced to rollback forever, even though no additional failures occur. This livelock problem must be solved by any rollback recovery algorithm.

7) **Message loss**: Recovery from a set of consistent checkpoints may cause message losses as illustrated in Section 2.3.

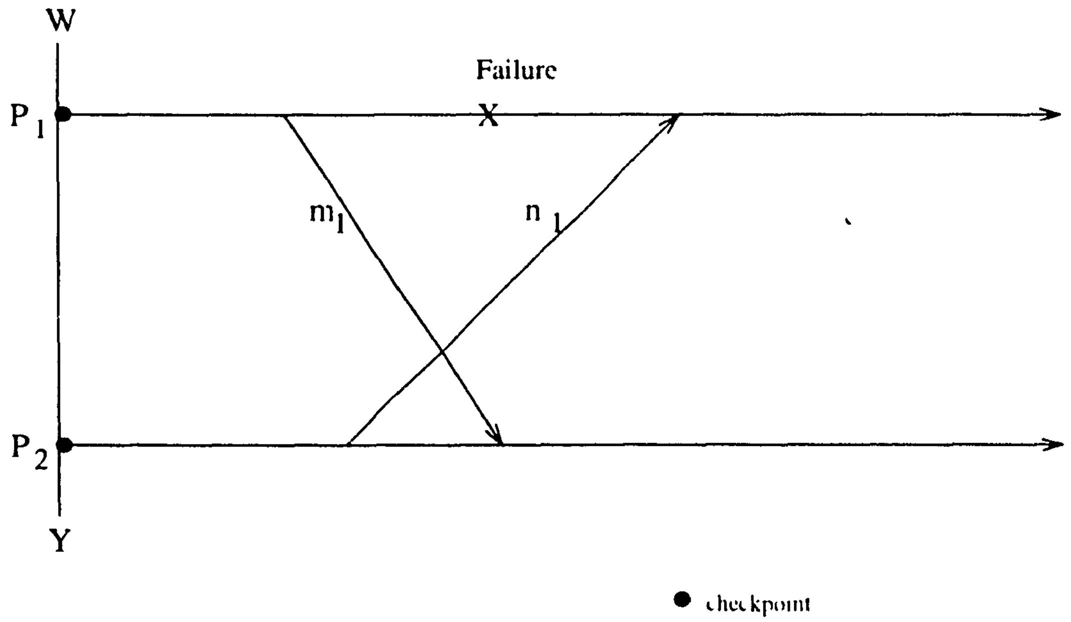


Fig. 2.7a. Histories of P_1 and P_2 up to P_1's failure

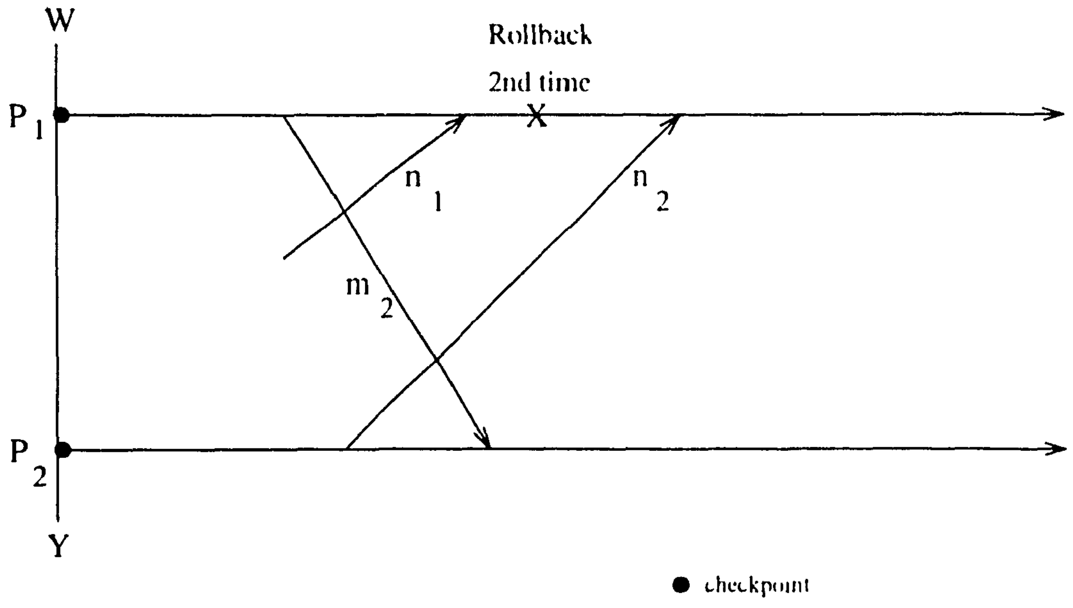


Fig. 2.7b. Livelock problem: histories of P_1 and P_2 up to P_1's second rollback

2.4. Classification of Checkpointing and Rollback Recovery Schemes

Distributed checkpoint and rollback recovery algorithms [ChLa85, JoZw90, KoTo87, LeBh88, SiWe89, StYe85, VRL87] can be classified into two categories according to the method used to create checkpoints with respect to recovery lines:

* **Pre-planned**: processes coordinate their checkpointing actions such that each process saves only its most recent checkpoints, and the set of checkpoints in the system is guaranteed to be consistent. The responsibility of obtaining the recovery points lies with the checkpointing algorithm rather than the rollback and recovery algorithm. When a failure occurs, the system restarts from these checkpoints.

* **Un-planned**: processes take checkpoints independently according to its own needs and without any synchronization with other processes. Multiple checkpoints have to be kept in local stable storage. Upon a failure, the rollback and recovery algorithm must find a consistent set of checkpoints among the saved ones (i.e., to select checkpoints such that they form a recovery line). The system is then rolled back to and restarted from this set of checkpoints. To aid in recovery and to minimize the amount of work undone in each process, messages that have been sent or received by a process are saved with the process state when a checkpoint is created [JoZw90, StYe85]. Algorithms that take this approach can be further classified into those that use pessimistic and those that use optimistic message logging.

In pessimistic (or synchronous) message logging, every message received is logged to stable storage before it is processed [BBG83, PoPr83]: a receiver process is blocked until the message is logged on stable storage. Thus the stable information across processes is always consistent. However, this method slows down every step of the application computation, because of the synchronization needed between logging and processing of incoming messages. On the other hand, in optimistic (or asynchronous) message logging, messages received by a process are logged in stable storage asynchronously from processing [ElZw92, JoZw90, SiWe89, StYe85]. In this case, logging can lag behind processing, a receiver process does not block to log each message. Failure-free computation is not disturbed, but some extra work must be done upon recovery to make sure that the restored states are consistent.

Pre-planned checkpointing schemes are always guaranteed to find recovery lines which are valid since the checkpointing of individual processes is synchronized such that the collection of checkpoints represent a consistent state of the whole system. In some un-planned checkpointing schemes, on the other hand, it cannot be guaranteed that checkpoints form part of any recovery line since these recovery lines must be extrapolated at the time of rollback. Consequently, un-planned strategies that cannot guarantee consistent recovery lines cannot ensure that a rollback will not necessitate a re-execution of the entire system.

These methods achieve only a subset of the goals of reducing the overhead during failure-free operation and limiting the extent of rollback. The overhead during failure-free computation includes message overhead between processes to ensure that a consistent system state is recorded to rollback to and the cost of saving the checkpoints on stable storage. The pre-planned strategies incur a higher overhead during the checkpointing phase since it is during the checkpointing phase that recovery lines are determined. The more infrequently the checkpointing computation is done, the more out-of-date the checkpoints will be, and thus the more work that will be lost (in spite of it being error-free) following a failure. The processors indeed rollback to a consistent state, but not necessarily to the maximum consistent state before the error event. If there are no failures, then the above approach places an unnecessary burden on the system in the form of additional messages and delays. On the other hand, un-planned strategies incur very little overhead during checkpointing but have a high overhead during the rollback phase. Since more than one process may be involved in the creation of checkpoints in the pre-planned schemes and the approach usually has a two-phase structure, the pre-planned schemes tend to introduce synchronization delays blocking the normal computation whereas in the un-planned strategies only a single task is involved in creating its own checkpoint.

Between the classes of the un-planned strategy, the main trade-off between the pessimistic and the optimistic approaches is the overhead during normal computation of logging the message against a more complex recovery. Faster recovery is achieved at the expense of greater run-time overhead or specialized hardware for pessimistic message-logging as compared to optimistic message-logging where messages not yet logged when a rollback is initiated can cause slower recovery.

2.5. Related Work in Checkpointing and Rollback Recovery Algorithms

Various checkpointing and recovery procedures have been introduced in the literature. Much of the previous work in checkpointing has focused on minimizing the number of processes that must participate in taking a consistent checkpoint or in rolling back [IsMo89, KoTo87, LeBh88]. Another issue that has received considerable attention is how to reduce the number of messages required to synchronize the consistent checkpoint [BCS84, TKT89, VRL87]. Researchers have succeeded in providing rollback by obtaining the recovery line joining the checkpoints of the processes immediately preceding the contaminated events. However, not much has been done in providing minimal rollback, which is the line joining the local states of the processes immediately preceding the events that may depend on the contaminated data produced as a result of the erroneous event. Moreover, most of the algorithms assume the processes to be fail-stop, i.e., the errors in a process are always detected immediately by the recovery system.

Under the schemes that fall into the pre-planned approach category:

Chandy and Lamport [ChLa85] have proposed a global snapshot algorithm where the checkpoints can be looked upon as valid recovery lines to which the system can roll back. Their algorithm is computationally expensive and does not provide minimal rollback. Also, it does not provide a meaningful state; the global state obtained may or may not be a global state through which the system has passed. To obtain an error-free checkpoint, no error should have occured in any of the processes between the state where the algorithm was initiated and the state where it was terminated. Also, after a loss of coordination between the processes of a protocol [Refer to Chapter 4], the application of distributed snapshot algorithm may or may not terminate; therefore, it is not guaranteed that it will return any global state [SUA92]. A modified distributed snapshot algorithm for the case when a loss of coordination occurs is given in [SUA94].

Koo and Toueg [KoTo87] give algorithms, also based on pre-planned approach, which guarantee that a minimal number of checkpoints will be created, and that a minimal number of processes will be forced to roll back should a failure occur. The obtained checkpoint is an efficient and meaningful state. Minimal storage requirements are needed since only the local states corresponding to checkpoints are stored. But the algorithms do not tend to consider the problem of message loss upon rollback recovery. However, they solve the livelock problem during recovery. They allow multiple rollback and checkpointing. One of the deficiencies of these algorithms is that they are intrusive; the processes are not able to exchange system application messages during checkpointing. There is a large system overhead, in the worst case of $3O(N^2)$ for each checkpointing and rollback recovery algorithm, where N is the number of processes. Their rollback and recovery algorithm does not provide minimal rollback. If each message transfer takes one time unit, then in the case of a fully-connected network and in the worst case when each process transmits system messages to all other processes after they take their last checkpoints, 5 time units are required for each checkpointing and rollback recovery procedure. Their checkpointing and rollback recovery algorithms are for fail-stop processes and are based on two-phase commit protocols.

The checkpoints created using the VRL algorithms [VRL87, Venk88] are always consistent. Only those messages that cause backward dependencies are stored as channel states in the checkpoint. The obtained checkpoint is an efficient and meaningful state. The algorithms are non-intrusive. They are inexpensive since the control information is only piggybacked onto the system messages during checkpointing. Of course, special control messages are required during rollback of the order of N^2 where N is the number of processes. Venkatesh *et al* claim that their strategy can be easily extended to support multiple rollbacks. The rollback recovery algorithm guarantees that minimal number of processes will be forced to roll back should a failure occur. However, their algorithm does not provide minimal rollback. It provides rollback to the most recent self-induced checkpoints (SIC) of the initiator process. It discards all checkpoints that follow it (without giving any procedure on how to do this) even though they may be used to obtain a consistent state close to the failure point. Rollback to the most recent SIC may still require non-erroneous events to be undone.

Israel and Morris [IsMo89] presented a non-intrusive checkpointing protocol that guarantees the existence of a globally consistent state, which is also close to the system state at the time of invocation. The protocol requires a minimal number of processes to checkpoint and is resilient to process failures. But they assume the processes to be fail-stop, i.e., the error should be detected as soon as it occurs and the failure should be detected by any live process. They also consider the system to be connected in a fully-connected configuration, which may not be true for many distributed systems. The algorithm cannot handle concurrent invocations. The number of checkpoint related messages is of the order of N where N is the number of processes in the system, and other related messages are of order $O(N^2)$. The number of time units required for the checkpointing protocol to terminate is $O(N)$.

Leu and Bhargava [LeBh88] present checkpointing and rollback algorithms that guarantee a globally consistent and meaningful state. The algorithms by themselves are intrusive. The non-intrusive version forces multiple invocations of the algorithms making the algorithms complex. The algorithms do not require all processes to checkpoint or rollback in case of failures. The algorithms allow concurrent invocations without any deadlock or livelocks. They handle multiple process failures during checkpointing or rollback of other processes. The processes not affected by error may need to rollback. The message overhead for checkpointing is not too high, of the order of three times the branches in the checkpointing tree. But an extra message exchange is involved during the derivation of the child-parent relationship. Similarly for the rollback operation the number of control messages required are $3N$ with extra messages required to establish rollback tree. The time for checkpointing and rollback operations is each (three times the depth of tree) time units.

Under the schemes that fall into the un-planned approach:

Strom and Yemini [StYe85] introduced the concept of optimistic message logging. They provide a consistent system state that is independent of orphan or lost messages. The time needed to create checkpoints is minimal and there is no overhead during checkpointing. However, they require a high overhead in time and an exponential number of message exchanges to recover from the failure of one processor. They require resending of application messages after recovery during replay, which will be discarded if those messages are already logged by the receiver process. The domino-effect is not completely eliminated. A processor rolls back $O(2^{|N|})$ times in the worst case where $|N|$ is the total number of processors. They have also not considered multiple failures of processes. The processes are assumed to be fail-stop.

Johnson and Zwaenepoel [JoZw90] find a consistent state from a set of independent checkpoints and messages logged onto the stable storage. Their algorithm provides minimal rollback if all the surviving information of all the surviving processes is logged. However, their algorithm is based on the assumption of processes being fail-stop. The recovery state algorithm can be used in recovering from any number of process failures in the system, including a total failure of all processes. The domino-effect is not completely eliminated if all messages received by each process are not eventually logged. A consistent state would not be obtained and earlier checkpoints would need to be considered to obtain a consistent state.

Sistla and Welch [SiWe89] have also presented algorithms to recover from crash failures. No further failures are allowed during the recovery procedure. O(N) extra information is appended to each application message and $O(N^2)$ messages are exchanged for rollback with no node rolling back more than once. They do not provide a fault-tolerant version of the recovery procedure. In their optimized use of log, there is a message loss problem since non-failed process may not roll back and resend messages, whereas the receiver process might have failed and lost the message or the receiver process considers it as an orphan and discards it. The algorithms do not also consider the problem of pre-rollback messages.

Elnozahy and Zwaenepoel [ElZw92] have provided an efficient rollback recovery protocol for a system of fail-stop processes. However, they have a complex recovery scheme and complex maintenance and storage of the antecedence graphs, with message overhead of $3N$ during recovery. There is no message loss problem and the pre-rollback messages are taken care by the use of incarnation numbers. The authors claim that the protocol also tolerates an arbitrary number of fail-stop failures, including additional failures during recovery. Their protocol does not provide minimal rollback.

Comparison and analysis of most of the algorithms presented in the literature can be found in [Agar94].

2.6. Concepts and Requirements for Checkpointing and Rollback

In a distributed system, processes perform computations on their respective local variables and send and receive messages to and from other processes. Events (sending or receiving messages) are assumed to be partially ordered by Lamport's "happened before" relation, and an event is uniquely identified within a process P_i by an event index EI_i. Each time a process receives or sends a message, it increments its event index EI_i by one.

The *maximally reachable event index tuple MREI(i)* [SaAg93a] for process P_i is denoted by $MREI(i) = (EI_1, EI_2, \cdots EI_i, \cdots EI_n)$, where $EI_1, EI_2, \cdots EI_n$ represent event indices that other processes must have reached before process P_i executes (sends or receives) its event with event index EI_i. This concept is based on the concepts of maximally executable sequence and maximally reachable state introduced in [Kak91] and the concept of event index as defined in [FoZw90], and is similar to the concept of vector clocks [RaSi96].

The maximum (minimum) of two maximally reachable event index tuples $MREI(i)$ of process P_i and $MREI(j)$ of process P_j (both processes belonging to the same system), denoted by $MAX(MREI(i), MREI(j))$ (MIN(MREI(i),MREI(j))), is a maximally reachable event index tuple whose kth element is a maximum (minimum) of the kth elements of $MREI(i)$ and $MREI(j)$.

Algorithm MREI specifies how to obtain a maximally reachable event index tuple, $MREI(i)$.

ALGORITHM MREI:

When a process P_i decides to send a message, it:

(M1) increments by one its local event index EI_i in $MREI(i)$,

(M2) tags the transmitted message with the maximally reachable event index tuple $MREI(i)$,

(M3) saves the corresponding executed event tuple (EXECEV tuple) in storage.

When a process P_j receives a message from process P_i along with $MREI(i)$, it:

(M4) increments by one its local event index EI_j in $MREI(j)$,

(M5) updates $MREI(j)$ such that $MREI(j) = MAX(MREI(i), MREI(j))$,

(M6) saves the EXECEV tuple in storage.

Using this algorithm, each process knows the maximally reachable event index of any other process after the execution of its events. Informally, the maximally reachable event index $MREI(i)$ represents the most recent local and consistent knowledge of the state of the global computation in the distributed system, from the point of view of process P_i.

In the following we show the proofs of correctness of Algorithm MREI [SAAA94]. We are mainly interested in showing that the maximally reachable event index tuple computed at each process represents a consistent view of the distributed system computation.

Lemma 2.1. The local event index EI_i at P_i in the maximally reachable event index tuple $MREI(i)$ is a consistent state of P_i.

Proof. This can be proved by contradiction. Suppose that the event index in the maximally reachable event index tuple for P_i does not represent a consistent state for P_i, that is, according to the definition of consistent state, it represents a situation where a message was never sent by another process. Then there are two possibilities: 1) the storage is corrupted, which is not possible according to our assumption about storage, or 2) the channel has generated the received message, which is also not possible because of our assumption about the reliability of the channel and its state on startup. Hence, the event index for P_i in EI_i represents a situation in which either P_i received a message from some other process, or P_i itself transmitted a message (according to the process specification). These two situations represent a consistent state. Hence, the event index EI_i for P_i in $MREI(i)$ represents a consistent local state for P_i.

Lemma 2.2. The event indices for all processes P_x in $MREI(i)$ provide consistent views of the states of their respective processes P_x, for $1 \leq x \leq n$ and $x \neq i$.

Proof. The event index for P_j in $MREI(i)$ is consistent because it was either received directly from P_j or indirectly from another process that received EI_j from P_j itself, which is therefore consistent, because of our assumptions about the reliability of the channel. The event indices for all other processes P_k in $MREI(i)$ are also consistent because they are consistent in their respective local processes and were transmitted along with application messages directly or indirectly to P_i.

Lemma 2.3. The maximally reachable event index tuple $MREI(i)$ at P_i represents a consistent view of the states of the local processes of the distributed system.

Proof. This is true since according to Lemmas 2.1 and 2.2, each event index in the maximally reachable event index tuple $MREI(i)$ represents a consistent state in a process.

Theorem 1. $MAX(MREI(i), MREI(j))$ represents a consistent view of the states of the processes in the distributed system.

Proof. According to the previous lemmas, the two maximally reachable event index tuples represent two consistent views of the states of the local processes. According to the algorithm, this maximum will be computed at P_i, therefore, the ith element of the maximum is always consistent and will be equal to P_i's view, because P_i will have the most up-to-date knowledge about its own index. The jth element of the maximum is also always consistent and is equal to the jth event index in $MREI(j)$, because P_j will have the most up-to-date knowledge about its own index. Finally, for the rest of the event indices, the maximum tuple will contain the maximum of two consistent views of the states of the processes. We are interested in the most up-to-date view for each process, which is therefore definitely consistent.

2.7. Summary

In this Chapter we have introduced the distributed system model based on the space time diagram and system interaction diagram. The basic concepts and definitions for fault-tolerance based on checkpointing and rollback recovery are reviewed. We also mentioned certain requirements that an ideal checkpointing procedure or an ideal recovery procedure should meet to provide efficient restart. Any checkpointing and recovery procedure should also solve the problems of pre-rollback messages, domino-effect, message-loss, orphan messages and livelock.

We have also presented an analysis of many important algorithms from the literature for checkpointing and rollback recovery in distributed systems. Most of the algorithms presented in this chapter have assumed the processes to be fail-stop, i.e., the errors are always detected immediately by the recovery system. Therefore, an error which occurs in one process can never contaminate further checkpoints and states. However, if reliable error detection is not always guaranteed, then rollback to a particular checkpoint may not guarantee correct performance and may require further rollbacks until a real safe state has been reached. The algorithms presented in this chapter provide efficient consistent checkpoints for the systems to rollback. Most of the algorithms allow a minimum number of processes to checkpoint and roll back in case of failures, but do not provide minimal rollback to the state event just before the effect of the error is manifested. They do provide rollback to the last consistent checkpoint state.

In this Chapter we have also presented our concepts on which our checkpointing/rollback recovery algorithms for a distributed system of non-fail-stop processes will be based. The next Chapter will provide our proposed procedures for checkpointing and rollback recovery in the distributed systems environment that will include most of the features of ideal checkpointing and recovery.

CHAPTER 3

FAULT-TOLERANCE IN DISTRIBUTED SYSTEMS

3.1. Introduction

Distributed computer systems are becoming increasingly popular and are being employed for critical applications demanding high reliability. They are more complex than centralized systems, which increases the potential for system faults. The development of procedures to ensure fault-tolerance in real-time distributed systems has been addressed in recent years. A well-recognized approach to implement fault-tolerance is to develop generic checkpointing and rollback recovery procedures. In contrast to checkpointing and recovery procedures for non-distributed systems, the complexity of similar procedures for distributed systems results from the fact that we are dealing with distributed and real-time processors, in which both partial ordering of distributed event occurrences and timing requirements must be respected. The key performance considerations in this approach are the overhead when no failure occurs, the extent of rollback, the number of control messages required during rollback, the number of processes required to roll back, and recovery within the specified real-time requirements.

In the previous chapter we have introduced an algorithmic procedure for efficient collection of contextual information in distributed systems, which is based on the concepts of event indices and maximally reachable event index tuples. This information is appended to each of the transmitted application messages. In this Chapter we will introduce algorithmic procedures for checkpointing and rollback recovery in distributed systems based both on the pre-planned and the un-planned approaches. Each process uses the recorded contextual information to decide on a checkpoint and/or recovery point that is globally consistent with the rest chosen in the other processes.

3.2. Checkpointing and Rollback Recovery Based on Pre-planned Approach

In this Section we first propose checkpointing algorithms for distributed systems to obtain global checkpoints, which can be initiated by any process in the system or upon failure of one or more component processes as part of a backward recovery procedure. We also present recovery procedures that utilise these global checkpoints to provide minimal rollback after failures.

In the concepts for checkpointing and rollback recovery in Section 2.6, the executed event tuple (EXECEV tuple) for the sender process P_i corresponds to ($MREI(i)$, *local state*), which is stored on stable storage, and the EXECEV tuple corresponding to the receiver process P_j is ($MREI(j)$, *local state* , P_i, *message received*), which is also stored on stable storage. Consider the time diagram and the exchange of messages shown in Figure 2.1. In this example, when m_2 is received by P_1, it increments its own event index and updates its version of P_2's event index by computing MAX(1000,0210). It stores the EXECEV tuple as (1210, $S1, P_2, m_2$). Similarly, when P_1 sends message m_5, it increments EI_1 in $MREI(1)$= 1210 and the EXECEV tuple stored is (2210, $S2$), and so on.

3.2.1. Checkpointing Algorithms

In this Section we present two checkpointing procedures based on our concepts. Procedure I can be initiated by a single process or concurrently by any number of processes in the distributed system. The procedure does not block the sending and receiving of application messages. In this procedure, it is assumed that there are no process failures in the distributed system during the invocation of this procedure (i.e. this is a non-fault-tolerant version). Procedure II is the fault-tolerant version of Procedure I, where checkpointing is invoked by a single process or concurrently by two or more processes periodically or after their failures are detected.

3.2.1.1. Procedure I: non-fault-tolerant checkpointing

(A) Initiator Process

When one or more processes P_i decide to checkpoint, the procedure executes the following:

(1) It initializes the local arrays *initiator* and ACK_T to *false* and $[-1,-1,\cdots,-1]$, respectively:

```
Forall j = 1 to N except i
{  $initiator[j] = false$;
$ACK\_T[j] = [-1,-1,\cdots,-1]$;  }
```

(2) It updates $EI(i)$ to reflect the last checkpoint tuple:

```
$EI(i) = MAX(EI(i), EI(k))$; where $EI(k)$ is the last checkpoint tuple
```

(3) It sends the maximally reachable event index tuple $EI(i)$ with the checkpoint message (CHK) to every other process:

```
Forall j = 1 to N except i
    Transmit $CHK(EI(i))$ to $P_j$;
```

(4) If it receives $CHK(EI(j))$ before receiving $ACK(EI(i))$ from any other process P_j, it updates $EI(i)$, however if an acknowledgement is received, it updates the array ACK_T.

```
Forall j = 1 to N except i
{
    Receive msg from $P_j$;
```

```
        IF msg = $CHK(EI(j))$ {
            $initiator[j] = true$;
            $EI(i) = MAX(EI(i), EI(j))$; }
        ELSE IF msg = $ACK(EI(j))$ then $ACK\_T[j] = EI(j)$;
    }
(5) Forany j = 1 to N except i
    IF $initiator[j] = true$
        Forall k = 1 to N except i
            Transmit $ACK(EI(i))$ to $P_k$;
(6) Expect acknowledgements from the processes who initiated checkpointing or for
    which $ACK\_T[j] \neq EI(i)$:
    Forall j = 1 to N except i
        IF ($initiator[j] = true$) OR ($ACK\_T[j] \neq EI(i)$) {
            Receive $ACK(EI(j))$ from $P_j$;
            $ACK\_T[j] = EI(j)$; }
(7) If Forall j = 1 to N except i
    $ACK\_T[j] = EI(i)$
        Checkpoint using the local state corresponding to its event index $EI_i$ in
        $EI(i)$ from stable storage
(8) $EI(k) = EI(i)$ for next checkpointing
(9) end.
```

(B) Non-initiator Process

When process P_j, which does not decide to checkpoint, receives $CHK(EI(i))$ from P_i

(1) It initiates the local array *initiator* and *ACK_T* to *false* and $[-1,-1, \cdots ,-1]$, respectively:

```
Forall k = 1 to N except j
{ $intiator[k] = false$;
$ACK\_T[k] = [-1,-1, \cdots ,-1]$; }
```

(2) It sends the acknowledgement message for the maximally reachable event index tuple $EI(i)$ to every other process:

```
$ACK\_T[i] = EI(i)$;
$EI(j) = EI(i)$;
Forall k = 1 to N except j
    Transmit $ACK(EI(j))$ to $P_k$
```

(3) If it receives $CHK(EI(k))$ before receiving $ACK(EI(k))$ from any process P_k, it updates $EI(j)$, however if an acknowledgement is received, it updates the array ACK_T:

```
Forall k = 1 to N except j and i
{
    Receive msg from $P_k$;
    IF msg = $CHK(EI(k))$ {
        $initiator[k] = true$;
        $EI(j) = MAX(EI(i), EI(k))$; }
    ELSE IF msg = $ACK(EI(k))$ then $ACK\_T[k] = EI(k)$;
}
```

(4) It sends an acknowledgement message with the updated $EI(j)$ to all processes if there is a concurrent initiation and $ACK(EI(j))$ was not already sent:

```
Forany k = 1 to N except j
    IF ($initiator[k] = true$) AND $EI(j) \neq EI(i)$
        Forall m = 1 to N except j
```

```
            Transmit $ACK(EI(i))$ to $P_{m}$;
(5) Expect acknowledgement from $P_i$ if there are concurrent invocations:
    Forany k = 1 to N except j
        IF ($initiator[k] = true$) {
            Receive $ACK(EI(i))$ from $P_i$;
            $ACK\_T[i] = EI(i)$; }
(6) Expect acknowledgements from the processes other than $P_i$ who initiated check-
    pointing or for which $ACK\_T[k] \neq EI(j)$:
    Forall k = 1 to N except j and i
        IF ($initiator[k] = true$) OR $ACK\_T[k] \neq EI(j)$ {
            Receive $ACK(EI(k))$ from $P_k$;
            $ACK\_T[k] = EI(k)$; }
(7) If Forall k = 1 to N except j
    $ACK\_T[k] = EI(j)$
        Checkpoint using the local state corresponding to its event index $EI_j$ in
        $EI(j)$ from stable storage
(8) $EI(k) = EI(j)$ for next checkpointing
(9) end.
```

The reception of $ACK(EI(j))$ from a process P_j ensures that the checkpoint signal has been received by process P_j and that process P_j is now aware of the latest maximally reachable event index tuple as the checkpoint, i.e., P_j has not failed or has not yet invoked the checkpointing procedure. Furthermore, messages received by process P_i from every other process P_j, after process P_i's event index in the global checkpoint. which have an index less than or equal to process P_j's event index in the tuple, form the incoming channel content for P_i. If these messages have been received by P_i, they are already in the stable storage and can be obtained later for recovery purposes.

Note that any process can initiate Procedure I since each process has the maximally reachable event index tuple for every other process after the occurrence of any event (transmission or reception). Because the channels are assumed to be reliable, it is clear that every process receives the checkpoint signal and checkpoints after its reception, and therefore the procedure is guaranteed to terminate in finite time.

3.2.1.2. Example I

Consider again the example shown in Figure 2.1. The sequences of EXECEV tuples stored in stable storage are as follows:

For P_1: (1210, S1, P_2, m2); (2210, S2); (3732, S3, P_2, m8); (4732, S4); send CHK(4732); (5932, S5, P_2, m12)

For P_2: (0110, S1, P_3, m1); (0210, S2); (0310, S3); (2410, S4, P_1, m5); (2532, S5, P_3, m6); (2632, S6); (2732, S7); (4832, S8, P_1, m9); (4932, S9); receive CHK(4732) from P_1, start checkpointing algorithm

For P_3: (0010, S1); (0322, S2, P_4, m4); (0332, S3); (2642, S4, P_2, m7); (2652, S5); receive CHK(4732) from P_1, start checkpointing algorithm; (2664, S6, P_4, m11)

For P_4: (0311, S1, P_2, m3); (0312, S2); (2653, S3, P_3, m10); (2654, S4); send CHK(2654)

Suppose that the procedure is invoked concurrently by processes P_1 and P_4 and that the last recorded checkpoint was (2410). According to the procedure P_1 sends (4732) with the checkpoint signal and P_4 sends (2654) with its checkpoint signal. The new global checkpoint (GCP) obtained using the procedure I is therefore at

(4754). The message received by P_2 after $EI_2(GCP) = 7$, i.e., m9 from P_1, has $[EI_1 = 4] \leq [EI_1(GCP) = 4]$. Therefore m9 forms the incoming channel content for P_2. Similarly, message m11 is the incoming channel content for P_3. Message m12 is not the incoming channel content for P_1 since $EI_2 = 9$ is greater than $EI_2(GCP) = 7$.

3.2.1.3. Procedure II: Fault-Tolerant Checkpointing

The following fault-tolerant procedure considers the case when checkpointing can be invoked by a single process or concurrently by two or more processes periodically or after their failures are detected. It generalizes the non-fault-tolerant version of Procedure I that now deals with one or more process failures during or before the invocation of the checkpointing procedure. The variable array element $Failure[i]$ indicates to process P_j whether or not a process P_i has failed. Variable $Failure[i]$ is set to true by P_i if P_i invokes the checkpointing procedure after it fails.

(A) Initiator process

When one or more processes P_i decide to checkpoint:

```
(1) Forall j = 1 to N except i { $intiator[j] = false$;
    $Failure[j] = false$;
    $ACK\_T[j] = [-1, -1, \cdots, -1]$; }
(2) $EI(i)$ = current MREI tuple of $P_i$ for periodic checkpointing (or MREI tuple
    $EI(i)$ corresponding to the event before the occurrence of an error leading to pro-
    cess failure in case of checkpointing upon failure)
(3) $EI(k)$ = last checkpoint tuple
(4) CHECKPOINT($EI(i)$, $EI(k)$, $Failure[i]$)
(5) end.
```

After entering CHECKPOINT(_,_,_) the value of any element of *Failure* [] cannot be changed externally.

Procedure CHECKPOINT($EI(i)$, $EI(k)$, $Failure[i]$)

```
(1) IF ( $(Failure[i] = true)$ AND $(EI_i$ in $EI(i) < EI_i$ in $EI(k))$ )
        $EI(i) = MIN(EI(i), EI(k))$;
    ELSE $EI(i) = MAX(EI(i), EI(k))$;
(2) Forall j = 1 to N except i
    IF $(Failure[i] = false)$
        Transmit $CHK(EI(i))$ to $P_j$;
    ELSE Transmit $CHKF(EI(i))$ to $P_j$;
(3) Forall j = 1 to N except i
    Receive msg from $P_j$;
    IF msg = $CHK(EI(j))$
    {
        $initiator[j] = true$;
        IF (Forall k = 1 to N $Failure[k] = false$)
            $EI(i) = MAX(EI(i), EI(j))$;
        ELSE
            IF ( (Forany k = 1 to N $Failure[k] = true$) AND
            $(EI_k$ in $EI(j) > EI_k$ in $EI(i))$ )
                $EI(i) = MIN(EI(i), EI(j))$;
            ELSE $EI(i) = MAX(EI(i), EI(j))$;
    }
    ELSE IF msg = $CHKF(EI(j))$
    {
        $initiator[j] = true$;
        $Failure[j] = true$;
        Forall k = 1 to N except j
            IF ( $EI_j$ in $EI(j) < EI_j$ in $EI(i)$ ) OR (($Failure[k] = true$) AND
            ($EI_j$ in $EI(j) > EI_j$ in $EI(k)$))
                $EI(i) = MIN(EI(i), EI(j))$;
            ELSE $EI(i) = MAX(EI(i), EI(j))$;
    }
    ELSE IF msg = $ACK(EI(j))$
        $ACK\_T[j] = EI(j)$;
(4) Forany j = 1 to N except i
    IF $initiator[j] = true$
        Forall k = 1 to N except i
            Transmit $ACK(EI(i))$ to $P_k$;
(5) Forall j = 1 to N except i
    IF ($initiator[j] = true$) OR ($ACK\_T[j] \neq EI(i)$)
        { Receive $ACK(EI(j))$ from $P_j$;
        $ACK\_T[j] = EI(j)$; }
(6) If Forall j = 1 to N except i
    $ACK\_T[j] = EI(i)$
        Checkpoint using the local state corresponding to its event index $EI_i$ in
        $EI(i)$ from stable storage
(7) $EI(k) = EI(i)$ for next checkpointing
(8) end.
```

(B) Non-initiator Process

When process P_j, receives $CHK(EI(i))$ / $CHKF(EI(i))$ from P_i

```
(1) Forall k = 1 to N except j
        $initiator[k] = false$;
        $Failure[k] = false$;
        $ACK\_T[k] = [-1, -1, \cdots, -1]$;
(2) IF $CHKF(EI(i))$ received $Failure[i] = true$;
(3) IF $Failure[j] = true$ /* due to the process $P_j$ interrupt mechanism */
        { $EI(j)$ = MREI tuple corresponding to the event before the occurrence of
        error leading to process $P_j$ failure;
        CHECKPOINT($EI(j)$, $EI(i)$, $Failure(j)$);
        STOP; }
    ELSE {
        EI(j) = EI(i);
        Forall k = 1 to N except j
            Transmit $ACK(EI(j))$ to $P_k$; }
(4) Forall k = 1 to N except j and i
    Receive msg from $P_k$;
    IF msg = $CHK(EI(k))$
    {
        $initiator[k] = true$;
        IF (Forall m = 1 to N $Failure[m] = false$)
            $EI(j) = MAX(EI(j), EI(k))$;
        ELSE
            IF ( (Forany m = 1 to N $Failure[m] = true$) AND
            ($EI_m$ in $EI(k) > EI_m$ in $EI(j)$) )
                $EI(j) = MIN(EI(j), EI(k))$;
            ELSE $EI(j) = MAX(EI(j), EI(k))$;
    }
    ELSE IF msg = $CHKF(EI(k))$
    {
        $initiator[k] = true;$
        $Failure[k] = true;$
        Forall j = 1 to N except k
            IF ( $EI_j$ $in$ $EI(k) < EI_k$ $in$ $EI(j)$ ) OR (($Failure[j] = true$) AND
            ($EI_k$ $in$ $EI(k) > EI_k$ $in$ $EI(j)$))
                $EI(j) = MIN(EI(j), EI(k));$
            ELSE $EI(j) = MAX(EI(j), EI(k));$
    }
    ELSE IF msg = $ACK(EI(k))$
        $ACK\_T[k] = EI(k);$
(5) Forany k = 1 to N except j
    IF ($initiator[k] = true$) AND ($EI(j) \neq EI(i)$)
        Forall m = 1 to N except j
            Transmit $ACK(EI(j))$ to $P_m$;
(6) Forany k = 1 to N except j
    IF ($initiator[k] = true$)
        { Receive $ACK(EI(i))$ from $P_i$;
        $ACK\_T(i) = EI(i);$ }
(7) Forall k = 1 to N except j and i
    IF ($initiator[k] = true$) OR ($ACK\_T[k] \neq EI(j)$ )
        { Receive $ACK(EI(k))$ from $P_k$;
        $ACK\_T[k] = EI(k);$ }
(8) If Forall k = 1 to N except j
    $ACK\_T[k] = EI(j)$
        Checkpoint using the local state corresponding to its event index $EI_j$ in
        $EI(j)$ from stable storage
(9) $EI(k) = EI(j)$ for next checkpointing
(10)end.
```

The value of Failure[j] cannot be changed between Steps 4 to 9. If a failure of process j occurs during Steps 4 to 9, the checkpointing algorithm is again initiated using (A) after completing all the steps of (B).

3.2.1.4. Example II

To illustrate Procedure II, we consider again the example in Figure 2.1. Suppose P_4 invokes the procedure after maximally reachable event index of (2654) with $Failure[4] = false$. Let the last checkpoint be the tuple (0310). Following the steps of Procedure II, different scenerios may occur:

Case(i). P_1 detects an error after its event index (4732) before receiving the CHK(2654) signal from P_4 that an error had occurred in P_1 after its event index of (2210). Then $EI(1) = MAX(2210,0310) = 2310$ and P_1 sends $CHKF$(2310) as the checkpoint upon failure signal. When P_4 receives the $CHKF$(2310), it sends an $ACK(MAX(2654,2310))$ and checkpoints when ACK(2654) is received from every other process. Similarly, other processes also checkpoint when ACK(2654) is received from every other process. Message m11 will be the incoming channel content for P_3 and the global checkpoint tuple is (2654).

Case(ii). If P_1 detects the error after receiving CHK(2654) from P_4 but before sending an ACK(2654), then $EI(1) = MAX(2654,2210)$ since $EI_1 = 2$ is not less than EI_1 received from P_4. It sends a $CHKF$(2654) to every other process and checkpoints when ACK(2654) is received from every other process. Similarly, other processes also checkpoint when *ACK*(2654) is received from every other process. Message m11 will be the incoming channel content for P_3 and the global checkpoint tuple is again (2654).

Case(iii). If P_1 detects the error after receiving *CHK*(2654) from P_4 but after sending an *ACK*(2654) to other processes. The global checkpoint as obtained by all processes after the termination of the procedure is (2654). Process P_1 initiates a new invocation of the checkpointing procedure with $EI(i) = 2210$ and $EI(k) = 2654$.

3.2.1.5. Correctness and Complexity

In this Section, we first prove the correctness of Procedure I and II. Then we describe their respective complexities.

Proofs of Correctness

Lemma 3.1. Every process terminates its execution of the checkpointing procedure.

Proof. Let P_i and P_j be the initiator processes that concurrently invoke the checkpointing procedure periodically or upon failure and P_k be the non-initiator process. When $P_i(P_j)$ receives the maximally reachable event index tuple from $P_j(P_i)$, it knows that some other process has initiated the procedure and is waiting for its acknowledgement, similar to the way $P_i(P_j)$ is waiting for an acknowledgement from every other process. So it takes the *MAX*(*MAX or MIN* in the case of the fault-tolerant version) of the two tuples and sends an *ACK* along with the *MAX*(*MIN*) tuple in the case of concurrent invocations. Therefore, $P_j(P_i)$ will not be in a deadlock waiting for an acknowledgement from other initiator processes. P_k sends an acknowledgement upon receiving the checkpointing signal from any of the processes P_i or P_j. But this acknowledgement may not be corresponding to the

MAX (*MIN*) tuple. When P_k receives *CHK* (*CHKF*) signal from the other process it sends an *ACK* along with the *MAX* (*MIN*) tuple. Therefore initiator processes will not wait for *ACK* (*MAX* /*MIN* tuple) from non-initiator processes also and will terminate when acknowledgement corresponding to the *MAX* (*MIN*) is obtained from all other initiator and non-initiator processes. Similarly non-initiator processes also receive *ACK* (*MAX* /*MIN* tuple) from all other initiator and non-initiator processes and will terminate.

Lemma 3.2. If the set of checkpoints in the system is consistent before the execution of the checkpointing Procedure I or II, the set of checkpoints in the system is consistent after the termination of either procedure.

Proof. If the set of checkpoints is consistent before the execution of the checkpointing procedure, then the maximally reachable event index tuple $EI(k)$ received during last checkpointing represents a consistent system state. Also, from the concepts of maximally reachable event index tuple as given in Section 2.6, $EI(i)$, which is the current maximally reachable event index tuple of P_i for periodic checkpointing or the tuple corresponding to the event before the occurrence of error, represents a consistent system state. Therefore the set of checkpoints obtained after the termination of the checkpointing procedure corresponding to the maximally reachable event index tuple $MAX(EI(i),EI(k))$ for non-fault-tolerant version and $MAX(EI(i),EI(k))$ or $MIN(EI(i),EI(k))$ for the fault-tolerant version, represent a consistent system state. For the fault-tolerant version the *MIN* function is applied to remove the dependencies of the events occuring after the failure.

Theorem 3.1. When Procedure I or II is applied, its termination is guaranteed after which the system reaches a consistent system state.

Proof. The proof is straightforward using Lemmas 3.1 and 3.2.

Complexity

The number of messages required for single invocation is $N(N-1)$, i.e., the message complexity is $O(N^2)$. For M $(M \le N)$ concurrent invocations, the number of messages required is $2N(N-1)$, which has again the complexity of $O(N^2)$.

The time required to perform checkpointing (if we assume that the message from one process takes $T = one$ unit of time to reach all of the other processes in a fully connected network system or $T = d$ units of time, where d is the depth of minimum spanning tree of a strongly connected network, to reach the leaf node of the spanning tree), is $2T$ time units for $M = 1$ or $M = N$ invocations and $3T$ time units for $1 < M < N$ invocations.

3.2.2. Rollback Recovery Algorithm

In this Section, the checkpointing procedures described in Section 3.2.1 will be used as the basis for optimal recovery in distributed systems requiring minimal rollback. In earlier work on recovery in distributed systems, a recovery point is defined to be the consistent state saved as a result of the last checkpointing procedure. However, in our approach, a recovery point is the state just before the occurrence of an error for the failed process, and the state just before the occurrence of the effect of that error for other processes. The recovery point may, in the worst case, be a state saved during checkpointing or a state further ahead in the transition sequence.

For the purpose of recovery we assume that there is a mechanism in the system to carry out error detection before checkpointing and that the checkpoint is a safe state. We also assume that all the states preceding the error in all the processes are correct (safe). A process also assumes that the received messages are correct and hence the process of fault-detection would not suspect the sender. Otherwise, each process will suspect the messages it receives and will throw the blame on some other process.

The storage required in our recovery algorithm is as follows. Each process P_i has a record on stable storage of the sequence of maximally reachable event index tuples $MREI_1(i), MREI_2(i), \cdots MREI_m(i), \cdots MREI_{lm}(i)$ that are known to that process after the last checkpoint along with their respective local process states. This can be obtained from the sequence of $EXECEV$ tuples as stored in Section 3.2.1.2. Also, for the purpose of recovery, the sender process stores in the $EXECEV$ tuples the id of the process to which the messages were sent. The new checkpoint is obtained after the event tuple $MREI_m(i)$, and $MREI_{lm}(i)$ corresponds to the last event which occurred before recovery. Moreover, a request to suspend transmission after checkpointing is sent along with the $CHKF$ signal.

The following recovery algorithm requires that only the affected processes perform minimal rollbacks. We assume that no failure would occur in the system during recovery.

3.2.2.1. Algorithm R1

(i) After process P_i receives a $CHKF(EI(j))$ signal from failed process P_j, and obtains the event indices corresponding to the global checkpoint $GCP = (CP_1, \cdots CP_j, \cdots CP_i, \cdots CP_n)$, it traces the recorded sequence of maximally reachable event index tuples, $MREI_1(i), MREI_2(i), \cdots MREI_x(i), MREI_{x+1}(i), \cdots MREI_{lm}(i)$, up to the tuple $MREI_{x+1}(i) = (EI_1, \cdots EI_j, \cdots EI_i, \cdots EI_n)$ where the event index EI_j of the failed process P_j is greater than the failed process's event index corresponding to the checkpoint CP_j (i.e., trace the sequence until when EI_j of

$MREI_{x+1}(i) > CP_j$). The recovery point RP_i for process P_i would then be EI_i of $MREI_x(i)$. If $MREI_{x+1}(i)$ cannot be obtained then EI_i in $MREI_{lm}(i)$ gives RP_i. The recovery point for the failed process P_j, of course, corresponds to CP_j.

(2) Process P_i sends RP_i to those processes to which messages were sent after the last checkpoint.

(3) The processes receive messages sent by other processes up to and including their recovery points. Recovery points are received from processes if their event indices in the message received were greater than their corresponding event indices in the last checkpoint.

(4) Consider the sequence of *EXECEV* tuples for the received messages:

```
For any process $P_m$ where $1 \le m \le N$:
IF $MREI_k(m) \le RP_k$ received from $P_k$
    If $MREI_m(m) \le RP_m$
        the stored message is not required to be replayed after recovery
    Else
        the message is replayed after recovery since this is the message sent by
        $P_k$ before its $RP_k$ and received by $P_m$ after its $RP_m$
Else remove message from storage since it is a contaminated message
```

(5) Remove the tuples received from the last checkpoint up to the recovery point to update the sequence of tuples to be stored:

```
Consider the sequence of EXECEV tuples for the received and sent messages:
For any process $P_m$ where $1 \le m \le N$:
    For tuples corresponding to sent messages:
        Remove the tuples from the sequence of EXECEV tuples since the
        message sent has either been received or was contaminated and sent
        after the recovery point
    Else retain the tuple in the sequence
```

For tuples corresponding to received messages:

If $MREI_m(m) > RP_m$

retain the message in the stored sequence

Else remove the message from storage since it has been received before its recovery point

The processes resume their executions from the states corresponding to their respective recovery points.

3.2.2.2. Example

To illustrate Algorithm R1, we consider the example shown in Figure 2.1 and reproduced in Figure 3.1. Let the last set of checkpoints correspond to the event indices (0310). Let process P_2 receive a $CHKF$ (2210) signal from P_1 and obtain the global checkpoint as (2310). It then traces the tuples recorded up to $MREI_{x+1}(2) = 4832$. Therefore, $MREI_x(2) = 2732$, and the recovery point $RP_2 = 7$ corresponding to $EI_2 = 7$ of $MREI_x(2)$. Process P_2 sends RP_2 to P_1 and P_3 and receives recovery point from P_3. Stored messages $m5$ and $m6$ are not required to be replayed. Message $m9$ is removed from storage since it is contaminated.

Similarly, P_3 finds its $RP_3 = 5$ and sends it to P_2 and P_4. It receives $m11$ from P_4 until recovery points are received from P_2 and P_4. Messages $m4$ and $m7$ do not need to be replayed. Message $m11$ is replayed since it is sent by P_4 before its recovery point and received by P_3 after its recovery point.

Process P_4 also finds its $RP_4 = 4$ and sends it to P_3. It receives $RP_3 = 5$ from P_3 and decides not to replay message $m10$.

Process P_1 receives message $m12$ before receiving $RP_2 = 7$ and decides to remove and discard it from storage.

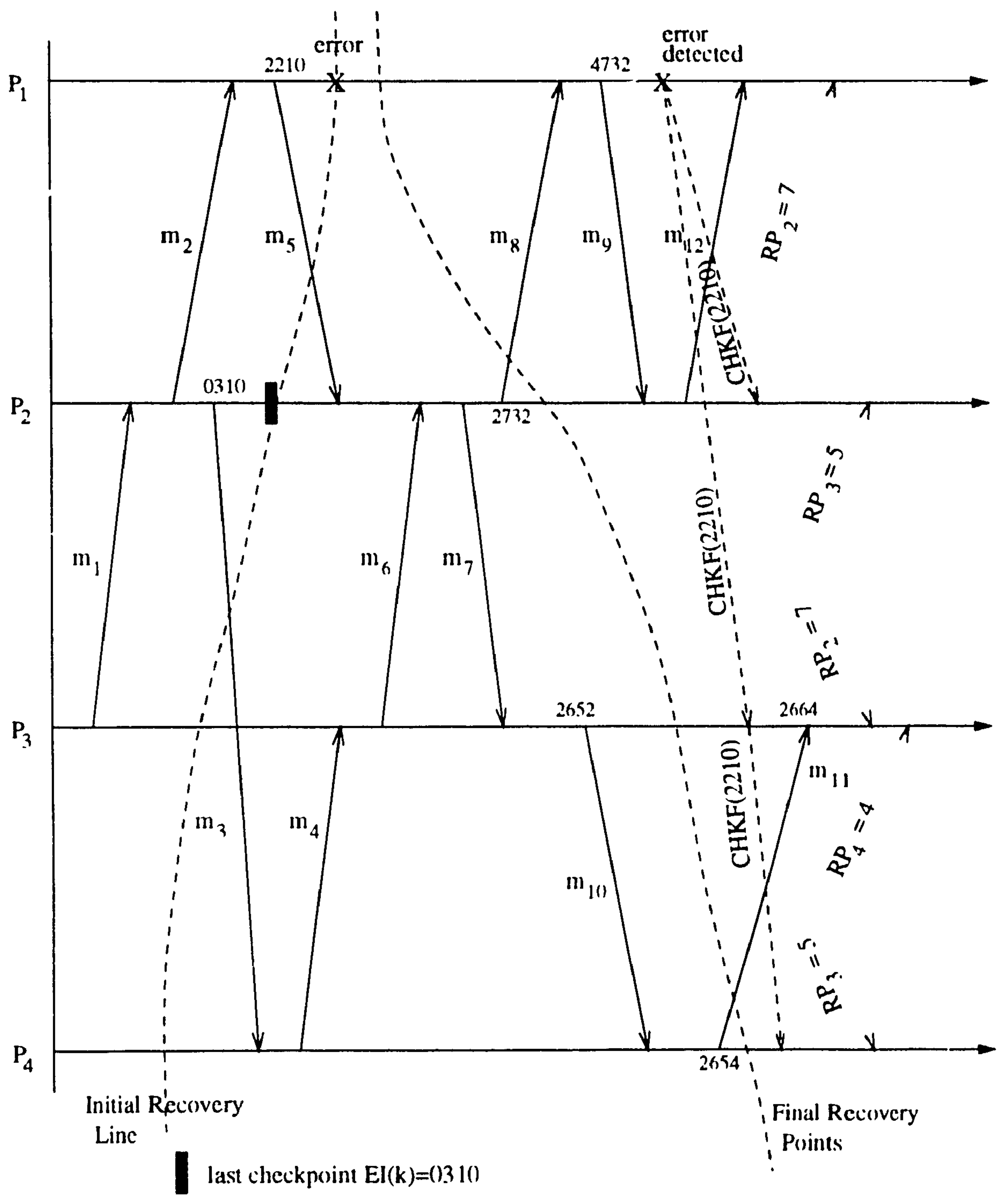


Figure 3.1. An Example to Illustrate Algorithm R1

After recovery the tuples retained in storage are:

For P_1: (3732, S3, P_2, m8);

For P_2:

For P_3:

For P_4:

The number of messages required for each non-failed process is equal to the number of processes to which messages were sent since the last checkpoint, and in the worst case is equal to (N-1)(N-1), i.e., of the $O(N^2)$. The time required is T time units where T = depth of minimum spanning tree.

3.2.2.3. Algorithm R2

For this algorithm we assume that a system failure can occur during the execution of the rollback procedure.

```
(1) If process $P_i$ fails (detects an error) during the execution of Algorithm R1, then:
        If $EI_i$ in the maximally reachable event index tuple $MREI(i)$ corresponding
        to the event before the ocurrence of error is $> RP_i$
            Continue the recovery algorithm R1
        Else
            exit recovery algorithm R1,
            discard any $RP_j$, if received, from any process $P_j$, and
            follow the appropriate checkpointing algorithm
(2) If process $P_j$ receives a $CHKF$ signal from process $P_i$ before resuming its execu-
    tion from the state corresponding to its $RP_j$, then:
        exit the recovery algorithm R1,
        discard any $RP_k$, if received, from any process $P_k$, and
```

apply the appropriate checkpointing algorithm

3.3. Procedures for Fault-Tolerance Based on the Un-Planned Approach

In this Section, the problem of fault-tolerance in distributed system is addressed by providing efficient algorithmic procedures for recovery in such systems. Our recovery procedures do not require the application of an intrusive checkpointing procedure, but use contextual information exchanged between the distributed system components during normal system progress. On detection of a failure, each process through an efficient propagation mechanism, will locally compute a recovery point that is mutually consistent with the recovery points computed by other processes.

Two approaches are considered for the storage requirement needed to implement our recovery procedures. These are described next.

3.3.1. Stable Storage Requirement for Message Logging

In the concepts for checkpointing and rollback recovery in Section 2.6, the executed event tuple (EXECEV tuple) for the sender process P_i corresponds to $(P_j, MREI(i), \textit{message sent}\ P_j)$. The EXECEV tuple corresponding to the receiver process P_j is $(P_i, MREI(j), \textit{message received}\ P_i)$. A record of the EXECEV tuples is appended to the process' execution history stored on the stable storage after every transmission or reception of a message. Consider again the time diagram and the exchange of messages shown in Figure 2.1. When m_2 is received by P_1, it stores the EXECEV tuple as $(P_2, 1210, m_2)$ on stable storage. Similarly, when P_1 sends message m_5, it stores $(P_2, 2210, m_5)$ on stable storage and so on.

3.3.1.1. Recovery Procedure

For the purpose of recovery, we define the in-neighbor (out-neighbor) of a process, say P_k, to be the set of processes from (to) which P_k can receive (send) application messages according to the distributed system specifications. For example, the in-neighbor and out-neighbor sets of process P_2 of the distributed system example of Figure 2.1 can be easily determined from its system interaction diagram in Figure 2.2 as $\{P_1, P_3\}$ and $\{P_1, P_3, P_4\}$ respectively.

The following recovery algorithm requires only the affected processes to perform minimal rollbacks on their respective computations. We assume the processes to be fail-stop. The recovery procedure provides us with a global consistent state before the event an error message is sent in the failed process and before the effects of the error reach other processes. We also assume that no failure would occur in the system during recovery.

(A) Initiator Process

When a process P_j detects an error caused by the reception of a message emanating from process P_i, it initiates the recovery procedure in which process P_j executes the following steps:

(1) It suspends transmission of application messages and it computes the maximum of the event index tuples as $MAX(MREI(i), MREI(j))$, where $MREI(j)$ is the event index tuple at P_j before receiving the erroneous message and $MREI(i)$ is the event index tuple sent along with the erroneous message. It then decrements by one EI_i in the maximally reachable event index tuple obtained. The recovery point RP_j for process P_j is EI_j in $MREI(j)$.

(2) It broadcasts a recovery control message (rcm) containing the tuple obtained in step (1) as the recovery point (RP) tuple and the i.d. of the process P_i to each of

the processes in *out-neighbor* $(P_j) \cup P_i$. The reason for including P_i is to let the erroneous process know as soon as possible about the occurrence of error, which will consequetly reduce the amount of rollback caused by the possibility of the circulation of more contaminated messages.

(3) It receives messages sent by all processes P_k, such that $P_k \in$ *in-neighbor* (P_j), until it receives a recovery control message from each process in *in-neighbor* (P_j).

(4) At this point, it restarts from the local state corresponding to RP_j and it first considers each of the messages recorded in the process' execution history whose event index in the associated tuple is greater than RP_j.

Let EI_k be the event index corresponding to process P_k (from which the message has been received) in the tuple associated with the recorded message.

Let RP_k be the recovery point for process P_k.

```
IF $EI_k \leq RP_k$
THEN
    - replay reception of the same message
ELSE
    - the message is removed from storage because it is contaminated and the
      process must wait for a replacement message to arrive from the same source
      process.
```

(5) It updates the storage for the sequence of EXECEV tuples corresponding to recorded messages by removing the tuples stored up to the recovery point RP_j. Such tuples have $EI_j \leq RP_j$.

(B) Other Processes

For processes other than P_j: When a failure is detected by a process, say P_j, all processes will eventually know of its occurrence. A process, say P_k, knows about the failure either directly, by receiving an RP tuple with the i.d. of the failed process P_i from process P_j (i.e., $P_k \in out{-}neighbor(P_j)$), or indirectly as described below. In either case, when process P_k receives an RP tuple, it executes the following steps:

(1) It suspends transmission of application messages and it traces the recorded sequence of maximally reachable event index tuples $MREI(k)_1.MREI(k)_2 \cdots MREI(k)_m$ up to the tuple $MREI(k)_{x+1}$ where EI_i of the failed process P_i is greater than EI_i in the RP tuple. RP_k would then be EI_k of $MREI(k)_x$. If $MREI(k)_{x+1}$ do not exist, then RP_k would be EI_k of $MREI(k)_m$.

(2) It sends a recovery control message containing an updated recovery point tuple (in which RP_k is the one found in the step above) and the i.d. of the failed process to the processes in $out{-}neighbor(P_k)$.

(3) It receives messages sent by all processes P_m in a temporary buffer, such that $P_m \in in{-}neighbor(P_k)$, until it receives a recovery control message from each process in $in{-}neighbor(P_j)$ except for the process(es) that have already sent a recovery control message.

(4) At this point, it restarts from the local state corresponding to RP_k and it first considers each of the messages recorded in the P_k's execution history whose event index in the associated tuple is greater than RP_k.

Let EI_m be the event index corresponding to process P_m (from (to) which the message has been received (sent)) in the tuple associated with the recorded message.

Let RP_m be the recovery point for process P_m.

```
IF the recorded message corresponds to a reception
THEN
    IF $EI_m \le RP_m$
    THEN
```

```
            - replay reception of the same message
        ELSE
            - the message is removed from storage because it is contaminated and
              the process must wait for a replacement message to arrive from the
              same the source process.
    ELSE /* if the recorded message corresponds to a transmission */
        - reexecute the transmission transition
```

(5) It updates the storage for the sequence of EXECEV tuples corresponding to recorded messages by removing the tuples stored up to the recovery point RP_k. Such tuples have $EI_k \le RP_k$.

The local states corresponding to their respective recovery points are also saved in stable storage for reference during further failures.

3.3.1.2. Example

To illustrate the recovery procedure consider the example of Figure 2.1 and reproduced in Figure 3.2. The failure is detected by process P_1 when message m_8 is received from process P_2. The sequence of EXECEV tuples recorded on stable storage before the detection of the error are as follows:

For P_1: (P_2, 1210, m2); (P_2, 2210, m5); error message received from P_2 with $MREI$(2)=2732; start recovery procedure

For P_2: (P_3, 0110, m1); (P_1, 0210, m2); (P_4, 0310, m3); (P_1, 2410, m5); (P_3, 2532, m6); (P_3, 2632, m7); (P_1, 2732, error message); receive recovery control message from P_1

For P_3: (P_2, 0010, m1); (P_4, 0322, m4); (P_2, 0332, m6); (P_2, 2642, m7); (P_4, 2652, m10); receive recovery control message from P_2

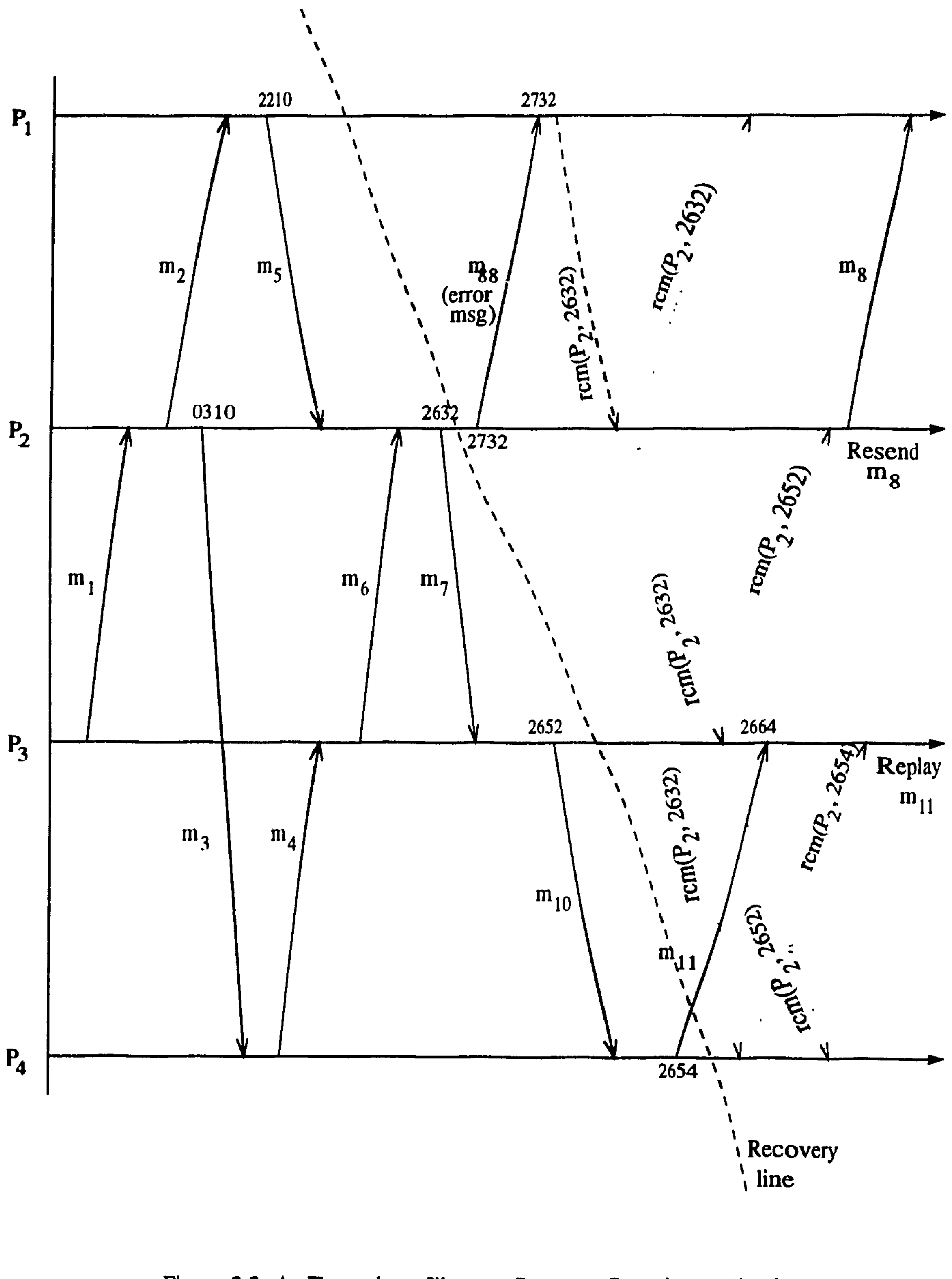


Figure 3.2. An Example to Illustrate Recovery Procedure of Section 3.3.1.1.

For P_4: (P_2, 0311, m3); (P_3, 0312, m4); (P_3, 2653, m10); (P_3, 2654, m11); receive recovery control message from P_2

Because P_1 detected the failure, it initiates the recovery procedure by suspending transmission of application messages and computing the MAX(2732, 2210) as (2732). It decrements EI_2 by one to give the resulting tuple as (2632). The recovery point RP_1 is therefore found to be 2. It then broadcasts a recovery control message containing (P_2, 2632) to each of the processes in its out-neighbor, i.e., to P_2. No message is sent by its in-neighbors. After its receives a recovery control message from P_2 it removes the original error message received and waits for the correct m_8 message to arrive. The stable storage is updated by removing the tuples up to the recovery point. Thus storage consists of the tuple with message m_8 for further failures.

Process P_2, after receiving recovery control message from P_1, suspends transmission of its application messages and traces the recorded history up to the tuple (2732). The recovery point RP_2 is therefore 6. It then sends the updated recovery control message containing (P_2, 2632) to its out-neighbors, P_1, P_3 and P_4. No message is received from any of its in-neighbors. After a recovery control message is received from P_3, it reexecutes from its last local state saved up to its recovery point and then resends message m_8. The stable storage is updated by removing the tuples up to its recovery point.

Similarly process P_3 finds its recovery point to be 5. Since it receives m_{11} before receiving the recovery control message from P_4, it saves it in a buffer to be considered later to be saved in stable storage. Thus after updating, P_3's stable storage contains the local state corresponding to its recovery point of 5 and reception tuple for message m_{11}.

Process P_4 finds its recovery point corresponding to its event of 4 and the stable storage after restarting contains the local state corresponding to event 4.

The recovery line corresponds to the recovery index in each of the processes and is (2654) with message m_{11} saved in stable storage. Only two processes are required to roll back (P_1 and P_2) and message m_{11} is to be received from buffer storage. Also, seven recovery control messages are needed to execute the recovery procedure. Figure 3.3 shows the recovery tree for our example, which depends on the in-neighbor and out-neighbor sets for each process and the process that detects the failure. The depth of the tree corresponds to the time units needed for recovery and is equal to 3 time units if the network topology is same as the system interaction diagram of Figure 2.2.

3.3.1.3. Correctness and Complexity

In the following we show the proofs of correctness of the recovery procedure. We are mainly interested in showing that no message loss, message duplication, or orphan message will be introduced during the recovery process and therefore proper termination of the procedure will be guaranteed.

Proofs of Correctness

Lemma 3.3. Every message reception replayed by one process will not be retransmitted by the sender process since the message has already been recorded on stable storage by the receiver process. Therefore, the message duplication problem is avoided during recovery.

Proof. Let $(P_k, MREI(j), message\ m\ received\ P_k)$ be a record of a message reception in P_j's history. According to our recovery procedure, a message replay will occur if: (1) $MREI_k(j) \leq RP_k$ (step 4 in the recovery procedure). In this case, we

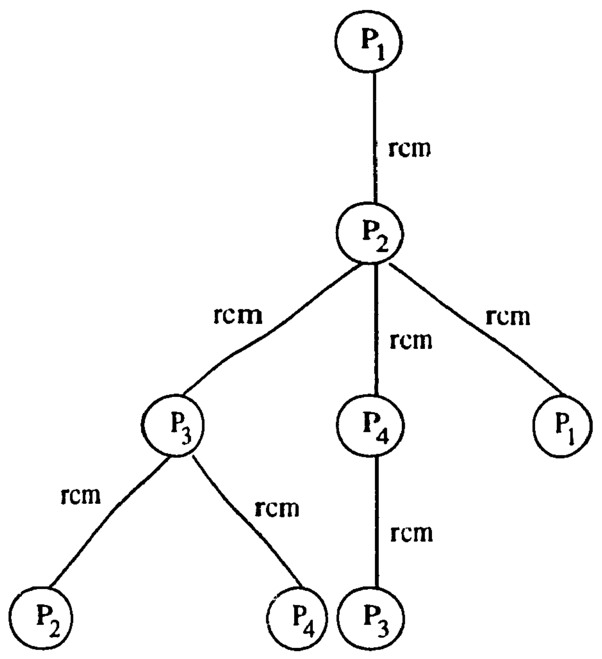


Figure 3.3. Recovery Tree for the Example of Section 3.3.1.2.

must show that the transmitter, say P_k, has recorded $(P_j, MREI(k), message\ m\ sent\ P_j)$ in which (2) $MREI_k(k) \leq RP_k$. But when message m is received we know from the procedure for updating $MREI(j)$ after reception of a message (step M5 of algorithm MREI in Section 3.6) that (3) $MREI_k(j) = MREI_k(k)$. By substituting (3) in (1), we obtain $MREI_k(k) \leq RP_k$, which is the condition for no retransmission in (2).

Lemma 3.4. Every message waited for by one process is going to be retransmitted by the sender process. Therefore no message loss is guaranteed and no deadlock will occur during recovery. This will guarantee progress during recovery.

Proof. Let $(P_k, MREI(j), message\ m\ received\ P_k)$ be a record in process P_j's history. According to our recovery procedure, m will be considered iff for a message m, $MREI_j(j) > RP_j$ (step 4 in the recovery procedure). The necessary condition that must hold in order for P_j to wait for retransmission of m is that (1) $MREI_k(j) > RP_k$. But we know from our algorithm MREI that when m is transmitted, (2) $MREI_k(k) \geq MREI_k(j)$. Using (1) and (2), we obtain $MREI_k(k) > RP_k$, which is the necessary condition for P_k to retransmit message m (step 4 in the recovery procedure).

Lemma 3.5. Every message retransmitted by one process is going to be waited for by another process. Therefore no orphan messages will be introduced during recovery.

Proof. Let $(P_j, MREI(k), message\ m\ sent\ P_j)$ be a transmission record in process P_k's history. According to our recovery procedure, message m corresponding to the above record will be retransmitted only if $MREI_k(k) > RP_k$. But P_j will wait for a retransmission iff (i) $MREI_k(j) > RP_k$ as received with the recovery control message of P_k. We already know from step 1 of the recovery procedure that $MREI_j(j) > RP_j$.

For message retransmissions, we have (1) $MREI_k(k) > RP_k$ and due to our algorithm MREI, we have (2) $MREI_k(j) = MREI_k(k)$ for message m. Substituting (2) in (1) we obtain $MREI_k(j) > RP_k$, which is condition (i) above, i.e., it is a message to be waited for.

Theorem 3.2. The recovery procedure is free from message duplications, message losses, and orphan messages.

Proof. The recovery procedure is free from orphan messages because every transmitted message is going to be waited for and expected (Lemma 3.5), and every message replayed by the receiver is not going to be considered for retransmission by the sender (Lemma 3.3). The procedure is free from deadlocks because every message waited for by the receiver is going to be retransmitted by the sender (Lemma 3.4).

Lemma 3.6. Every process P_i will eventually restart from its local state corresponding to RP_i.

Proof. According to our procedure, each process will move from the blocking state to the restart state only once a recovery control message has been received from each of its in-neighbors. Because the channels are supposed to be reliable, then eventually, after some finite time, all recovery control messages will reach their destinations. Therefore, according to step 4 of the initiator or other process, each of the processes will move to its restart state.

Theorem 3.3. The recovery procedure is guaranteed to terminate and the distributed system computations will eventually resume from a normal state.

Proof. Lemma 3.6 guarantees that the first phase of the procedure terminates properly, that is, each of the processes in the distributed system will eventually move from the blocking state to the restart state at which the process' history is traced back and each of the recorded events is considered. Also, Theorem 3.2 shows that the second phase in which histories are considered does not introduce orphan, duplicated, or missing messages and therefore each of the events recorded in the finite history will eventually be considered and the distributed system can resume normal execution.

Complexity

The maximum number of control messages required for recovery is Σ#(out-neighbor(P_x)), for every process P_x in the distributed system. An additional message would be needed (according to step 2 of initiator process) if the sender of the erroneous message does not belong to out-neighbor(P_j), where P_j is the initiator process. Moreover, if we suppose that each transmission takes one time unit, then the number of time units required for recovery is equal to the depth of the recovery tree, and the number of edges corresponds to the number of recovery control messages needed to implement the recovery procedure.

3.3.2. Volatile Storage Requirement for Message Logging

To recover from process errors and failures, and restore the system to a consistent state, we use two types of logs—a volatile log and a stable log. Accessing volatile logs requires less time, but the contents of a volatile log are lost if the corresponding processor is in an error state leading to its failure. At irregular intervals each process P_i independently saves its local state along with its *MREI*(i) tuple and incarnation number (*IN#*) in stable storage as its independent checkpoint *ICP*(i).

The initial state is also saved as a checkpoint by each processor. Stable storage is also required to save the global consistent checkpoint $GCP = (GCP_1, GCP_2, \cdots GCP_n)$ at the initial state of the system as (0000) and later as obtained by the recovery procedure after failures. In the recovery algorithm, only the $MREI(i)$ stored in an $ICP(i)$ is used. Once the restart point has been established, the process will be restarted using the $MREI(i)$ and the local state. The volatile log is used to save critical contextual information such as the EXECEV tuples which consists of the following:

For messages received: (incarnation number received, P_r (process from which message is received), $MREI(r)$ received, $MREI(i)$ obtained, message received)

For messages transmitted: (incarnation number sent, P_s (process to whom message is transmitted), $MREI(i)$ sent, message sent)

Incarnation numbers [StYe85] are used to distinguish messages sent before and after recovery. A new incarnation is started at the beginning of each recovery and the incarnation number is incremented by 1. Each message sent is also tagged with the current incarnation number of the sender. A record of the EXECEV tuples is appended to the process' execution history on the volatile log after every transmission or reception of messages. Consider the space-time model of a distributed computation shown in Figure 2.1. When m_2 is received by P_1, it stores the EXECEV tuple as (0, P_2, 0210, 1210, m2) on volatile storage. Similarly, when P_1 sends message m_5, it stores (0, P_2, 2210, m5) on volatile storage and so on. No messages are required to be logged to the stable storage during execution, thus reducing the overhead when no failures occur and the time overhead during recovery because the non-failed processes do not have to obtain the message log from stable storage.

3.3.2.1. Recovery Procedure

In this Section we present a rollback recovery algorithmic procedure for a distributed system of non-fail-stop processes, meaning that an error occuring in a process may not be detected immediately by the recovery system. Therefore, an error that occurs in one process can contaminate further checkpoints and states of itself and other processes in the system. Most of the existing recovery procedures either require the establishment of a global checkpoint by applying a checkpointing procedure periodically or the obtaining of a global consistent state during the recovery phase for fail-stop processes. Our concern here is the development of a minimal-time recovery procedure for non-fail-stop processes that can obtain most of the ideal requirements for recovery procedures listed in Section 2.3.1.

We do not discuss error detection here, but we do assume that there is a mechanism in the system to carry out error detection that is correct, such that there are no messages that were sent by one process after its error state that were received by another process before its error state, as shown in Figure 3.4, i.e., the error states detected in the system should be consistent. By this assumption we mean that all the states preceding the error in all the processes are correct. We also assume that once an error is detected by a process, the process will be able to initiate the recovery procedure. Our recovery algorithm can handle any number of processes detecting an error leading to its failure, including total failure of the system. However, we also assume that no failure occurs in non-failed processes in the system during recovery.

We first provide an informal description of our recovery procedure, then we formally describe the steps performed in the procedure, and finally we provide the proofs of correctness and the complexity of the procedure.

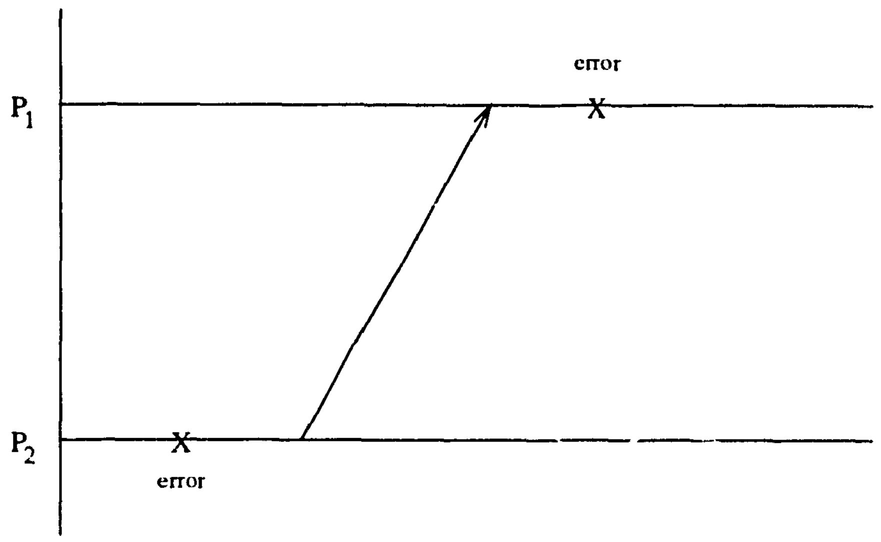


Figure 3.4. Inconsistent Error States

Informal description

Our recovery procedure does not require any intrusive pre-planned checkpointing procedure. Instead, independent checkpoints, $ICP(i)$, are saved in stable storage of every process P_i. A global consistent checkpoint is obtained by our recovery procedure considering the $MREI(i)$ corresponding to the independent checkpoints $ICP(i)$, the error event index tuple $EREI(i)$ (the $MREI(i)$ just before the error event for the faulty process P_i), and the effect of the error on non-faulty processes. Note here that the $MREI(i)$ within the $ICP(i)$ may not be the same as $EREI(i)$. If the current $MREI(j)$ for a non-faulty process P_j is such that it is not affected by the error, then it is not required to restart from its latest independent checkpoint. It only needs to resend messages to the failed processes, since the volatile storage of failed process is lost upon failure. If a non-faulty process is affected by the error, it re-executes from its independent checkpoint before the effect of the error, replaying messages from its volatile log and resending messages only to the failed processes, up to the event that is just before the one affected by the error. The event index tuple corresponding to this event is part of the global consistent checkpoint.

After detecting an error, the faulty process P_f sends a recovery message to all other processes in the system through its shortest path between the faulty process and the non-faulty process P_{nf}, and blocks any transmission of application messages. The non-faulty processes, after receiving the recovery message, block any transmission of application messages, find $MREI(nf)$ before the error event and send it with an acknowledgement to the faulty process. P_f then finds the global consistent checkpoint, GCP, from the information obtained from P_{nf}, and sends the GCP to all other processes. P_f then restarts from its recovery point corresponding to its independent checkpoint, withholding messages already sent to P_{nf} before its error event. Similarly, non-faulty processes after receiving the GCP from P_f restart from their recovery points, resending messages to P_f before the GCP. Messages to be replayed/received after the *GCP* are checked to see if they are contaminated and processed accordingly. Volatile storage for P_{nf} is made free of any events executed before the *GCP*.

The recovery procedure terminates after considering all the recorded events and saving the local states corresponding to the GCP and the contents of the channel, if any, in the stable storage. We shall prove later that our recovery procedure will terminate and is free from design errors. We show that every retransmitted message will be waited for, every waited for message will be retransmitted and finally every replayed message is not going to be retransmitted. These features amount to freedom from message loss, message duplication or orphan messages during the recovery process.

The Recovery Procedure

The community of processes divides into three parts: the faulty process, P_f ; the non-faulty process(es), P_{nf}; and any other process(es) that have also detected an error, P_{of}. We present first the algorithm executed by the faulty process(es), P_f (and P_{of}), and then the algorithm executed by the non-faulty process(es), P_{nf} .

The notation $ICP(i)$ is used in this section to denote one of the independent checkpoints recorded by process P_i. The notation $ICP_j(i)$ is used to denote the event index EI_j stored in the $MREI(i)$ contained with the $ICP(i)$. $EREI(i)$ is the error event index tuple, which is the $MREI(i)$ just before the faulty event for process P_i. $EREI_j(i)$ denotes the event index EI_j in position j of $EREI(i)$.

For faulty process, P_f

When a process P_f detects an error that had occurred after the event with event index tuple $EREI(f)$, it performs the following execution

(IP1) find the latest $ICP(f)$ such that $ICP_f(f) \le EREI_f(f)$, discard any checkpoints after this $ICP(f)$

(IP2) $IN\# = IN\# + 1$

(IP3) ** send recovery-message $RM(P_f, IN\#, ICP(f), EREI(f))$ to all other processes and wait for their $RMack(P_f, IN\#, MREI(nf))$ or $RM(P_{of}, IN\#, ICP(of), EREI(of))$ messages

** Application messages, with $IN\#$ not updated, received from all other processes before receiving $RMack$ or RM messages are discarded since they may be contaminated and will be resent in the ELSE part of (OP8) for non-failed processes.

** IF an $RM(P_{of}, IN\#, ICP(of), EREI(of))$ is received from any other failed process P_{of} instead of the expected $RMack(P_f, IN\#, MREI(nf))$ message THEN

    * find $ICP(f)$ such that $(ICP_{of}(f) \le ICP_{of}(of))$ AND $(ICP_f(f) \le ICP_f(of))$, discard any checkpoints after this new $ICP(f)$

    * receive $RMack(P_{of}, IN\#, MREI(nf))$ from non-failed processes

(IP4) After $RMack(P_f, \cdots)$ (or $RM(P_{of}, \cdots)$ plus $RMack(P_{of}, \cdots)$) messages are received from every other process

    ** find restart point for P_f, $RP(f) = ICP(f)$ where $ICP(f)$ is the last of all the M $ICP(f)$s, where M is the number of failed processes

    ** find the global consistent checkpoint tuple, GCP, such that

$$GCP_f = EREI_f(f)$$

$$GCP_{of} = EREI_{of}(of)$$

$$GCP_{nf} = MREI_{nf}(nf)$$

    where $MREI_{nf}(nf)$ is the last of the m $MREI_{nf}(nf)$ received in $RMack(P_{of}, IN\#, MREI(nf))$ and $RMack(P_f, IN\#, MREI(nf))$ messages from P_{nf}

```
(IP5)   save $GCP$ in stable storage
(IP6)   send $GCP$ and $RP(f)$ to all other processes
(IP7)   receive $GCP$ and $RP(of)$ from other failed processes
        (Note: $GCP$s received should be the same.)
(IP8)   Restart from the local state corresponding to its $RP_f(f)$
            ** Continue execution up to the event with $(MREI_f(f) \leq EREI_f(f))$
            AND $(MREI_{of}(f) \leq EREI_{of}(of))$ re-receiving messages with
            $(EI_f \leq GCP_f)$ AND ($IN\#$ received = its $IN\#$)
                - resend messages to $P_{of}$ with $EI_f > RP_f(of)$
                - do not resend messages to $P_{nf}$
            ** Save (local state, $MREI(f)$, $IN\#$) in stable storage
            ** Continue execution after $MREI_f(f)$
                - save in stable storage the messages received from $P_i$ that have
                $MREI_i(i) \leq GCP_i$
(IP9)   Remove $ICP(f)$s from stable storage with $ICP_f(f) < GCP_f$
```

For non-faulty processes P_{nf}

When process P_{nf} receives the recovery-message $RM(P_f, IN\#, ICP(f), EREI(f))$ from P_f, it performs the following algorithm

```
(OP1)   $IN\# = IN\#$ received
(OP2)   find the latest $ICP(nf)$ such that $ICP_f(nf) \leq EREI_f(f)$; discard any check-
        points after this $ICP(nf)$
(OP3)   IF the current $MREI_f(nf) \leq EREI_f(f)$
            THEN send $RMack(P_f, IN\#, MREI(nf))$ to $P_f$
            ELSE
            ** find in the tuples stored in volatile storage after $ICP(nf)$ the
            last EXECEV tuple that has $MREI_f(nf) \le EREI_f(f)$. Remove
            EXECEV tuples from volatile storage after this $MREI(nf)$. Now
            current $MREI(nf)$ is the same as this $MREI(nf)$
            ** send $RMack(P_f, IN\#, MREI(nf))$ to $P_f$
(OP4)   Wait for $GCP$ and $RP(f)$ from $P_f$
        ** messages received from all other processes before receiving $GCP$
        and $RP(f)$ from $P_f$ are stored temporarily in a buffer before being pro-
        cessed as in step (OP9)
        ** IF $P_{nf}$ receives $RM(P_{of}, IN\#, ICP(of), EREI(of))$ from any $P_{of}$
        before receiving $GCP$ and $RP(f)$ from $P_f$
            - send $RMack(P_f, IN\#, MREI(nf))$ to $P_{of}$
            - do steps (OP2) and (OP3) with the failed process as $P_{of}$
            - also send $RMack(P_{of}, IN\#, MREI(nf))$ to $P_f$
(OP5)   Receive $GCP$ and $RP(f)/RP(of)$ from all failed processes $P_f/P_{of}$
        (Note: $GCP$s received should be the same)
(OP6)   The restart point for $P_{nf}$, $RP(nf)$ is the $ICP(nf)$ found in step OP2 after
        receiving the last RM from any failed process
(OP7)   Consider tuples stored in volatile storage up to and including $RP(nf)$
        ** resend messages to the failed processes, $P_f$s, which have
        $(EI_f \le GCP_f)$ AND $(EI_{nf} > RP_{nf}(f))$
(OP8)   IF its current $MREI_{nf}(nf) = GCP_{nf}$
        THEN
        * no need to reexecute from $RP(nf)$
        * resend only those messages after $RP(nf)$ to failed processes $P_f$s that
        have $(EI_f \le GCP_f)$ AND $(EI_{nf} > RP_{nf}(f))$
        * save the current (local state, $MREI(nf)$, $IN\#$) in stable storage
    ELSE restart from the local state corresponding to its $RP_{nf}(nf)$
        * Consider tuples stored in volatile storage after $RP(nf)$ and having
        event indices with $EI_{nf} \le GCP_{nf}$
            - resend messages to failed processes that have ($EI_f \le GCP_f$)
            AND ($EI_{nf} > RP_{nf}(f)$)
            - consume the messages received
            - do not resend messages to other non-failed processes
        * Save the checkpoint (local state, $MREI(nf)$ corresponding to the
        event with $GCP_{nf}$, $IN\#$) in stable storage
(OP9) Continue the execution after $GCP_{nf}$
        * Consume from volatile storage the messages received from any pro-
        cess, $P_i$, with $MREI_i(i) \le GCP_i$ and save these messages in stable
        storage. Other messages are discarded from volatile storage
        * messages received in buffer/channel from any process $P_i$ with
        ($MREI_i(i) > GCP_i$) AND ($IN\# < itsIN\#$) are discarded
        * messages in buffer/channel from any process $P_i$ with
        $MREI_i(i) \le GCP_i$ are received and stored in stable storage
(OP10) Remove all $ICP(nf)$s having $ICP_{nf}(nf) < GCP_{nf}$ from stable storage
(OP11) Remove all EXECEV tuples having $MREI_{nf}(nf)$ in
    $MREI(nf)$ obtained $\le GCP_{nf}$ from volatile storage
```

3.3.2.2 Correctness and Complexity

The correctness of our recovery procedure can be proved by showing that eventually each non-failed process will receive the same maximum consistent global checkpoint from all failed processes and that each process will save the local state corresponding to the event index of its GCP and the contents of its input channels, if any, in the stable storage. In the following we shall prove that our recovery procedure will terminate and is free from design errors. We show that every retransmitted message will be waited for and every waited for message will be retransmitted and finally every replayed message is not going to be retransmitted. These features amount to freedom from message loss, message duplication or orphan messages during the recovery process.

Proofs of Correctness

Lemma 3.7. Every process will eventually restart from its local state corresponding to $RP_i(i)$ and will save the local state corresponding to the maximum consistent global checkpoint.

Proof. According to our procedure, each non-failed process P_{nf} will send "M" RMack messages to the "M" failed processes, and the last of the RMack message sent to all the failed processes will include the MREI of the non-failed process for which its $MREI_{nf}(nf)$ will be just before the error event indices for all failed processes [step OP3]. Since the channels are assumed to be reliable and FIFO, each failed process will receive the $MREI_{nf}(nf)$ in the last RMack message and will include that in the maximum consistent global checkpoint [step IP3]. Each P_{nf} will therefore receive the same GCP from each failed process and restart from its recovery point, replaying events from its volatile storage up to its GCP_{nf} before saving the local state corresponding to the maximum event free from any errors [step OP8]. The failed processes also restart from their recovery points up to the error event index, which is a part of the GCP, before saving the local state corresponding to the event before the error event [step IP7]. Any message that was received after its GCP but is free from error is saved in stable storage as channel contents [step IP7 and step OP9]. The local states corresponding to the GCP form a consistent state since no messages are replayed to be received that were sent after error event state.

Lemma 3.8. Every message replayed by non-failed processes P_j will not be retransmitted by the sender process P_i. Therefore the orphan message problem is avoided during recovery.

Proof. Let $(0,P_i,MREI(i),MREI(j),m)$ be a record in volatile storage of a message reception in non-failed process, P_j's history. According to our recovery procedure, a message m will be replayed by P_j if 1) $MREI_i(j) \le GCP_i$ [step OP8 and step OP9]. In this case, we must show that the transmitter, say P_k, does not retransmit message [steps IP7 and OP8] with 2) $MREI_k(k) \le GCP_k$. But when message m is received and because of the maximal event index principle, we have 3) $MREI_k(j) = MREI_k(k)$ and by substituting 3) in 1), we obtain the condition for no retransmission in 2).

Lemma 3.9. Every message waited for by a process is going to be retransmitted by the sender process. Therefore the message loss problem is avoided during recovery.

Proof. Let $(0,P_k,MREI(k),MREI(j),m)$ be a record in volatile storage of process P_j's history. According to our recovery procedure, m will be considered to be waited for iff for the message m, $MREI_j(j) > GCP_j$ [step OP8 and OP9]. The necessary condition that must hold in order for P_j to wait for retransmission of m is that 1) $MREI_k(j) > GCP_k$. But we know from the procedure for updating $MREI(j)$ after reception of a message (step M5 of algorithm MREI) that 2) $MREI_k(j) = MREI_k(k)$. Using 1) and 2), we obtain $MREI_k(k) > GCP_k$, which is the necessary condition for P_k to retransmit message m [steps IP7 and OP9].

Lemma 3.10. Every message retransmitted by one process is going to be waited for by another process. Therefore no duplicate messages will be introduced during recovery.

Proof. Let $(0, P_f, MREI(k), m)$ be a transmission record in process P_k's history. According to our procedure message m corresponding to the above record is retransmitted with updated IN# only if $MREI_k(k) > RP_k(f)$. But P_f's volatile storage is lost and it discards messages with IN# not updated [step IP3] and it waits for a retransmission of all application messages with updated IN# and i) $MREI_k(f) > RP_k(f)$. For message retransmission we have 1) $MREI_k(k) > RP_k(f)$. But because of the procedure for updating $MREI(j)$ after reception of a message (step M5 of algorithm MREI), we have 2) $MREI_k(f) = MREI_k(k)$. Substituting 2) in 1), we obtain $MREI_k(f) > RP_k(f)$, which is condition i) above. Thus m is going to be among those messages considered waited for by P_f, thus Lemma 3.10 is satisfied.

Theorem 3.4. The recovery procedure is free from message duplications, message losses, and orphan messages.

Proof. The recovery procedure is free from orphan messages because every message included before the GCP of a process will not be retransmitted by the sender (Lemma 3.8), and every message retransmitted by a process is going to be waited for and expected (Lemma 3.10). The procedure is free from deadlocks because every message waited for by the receiver is going to be retransmitted by the sender (Lemma 3.9).

Theorem 3.5. The recovery procedure is guaranteed to terminate and the distributed system computations will eventually resume from a normal state corresponding to the maximum consistent global state.

Proof. Lemma 3.7 guarantees that the procedure terminates properly, that is, each of the processes in the distributed system will eventually move from the wait state to the restart state at which the process' history is traced back and each of the recorded events is considered. Also, Theorem 3.4 shows that when histories are considered it does not introduce orphan, duplicate, or missing messages and therefore each of the events recorded in the finite history will eventually be considered after a finite time and the distributed system can resume normal execution.

Complexity

The maximum number of control messages required for recovery depends upon the number of processes detecting error in the system. For a single process in error $3(N-1)$ messages are required after error detection, where N is the total number of processes in the system, for rollback recovery and obtaining the maximum global consistent state. For a system where M errors are allowed ($M \leq N$), the number of control messages required is $2M(N-1) + (N-M)M^2$, which amounts to $2N(N-1)$ messages for a total failure. No communication overhead is involved during independent checkpointing. Moreover, if we suppose that each trasmission takes one time unit, the number of time units required for recovery is equal to three for a system connected in a fully connected network. In the case of a strongly connected network, the number of time units is three times the depth of the minimum spanning tree.

3.3.2.3. Example

To illustrate the recovery procedure, consider the example of Figure 2.1, with the independent checkpoints and recovery messages added as shown in Figure 3.5. The errors that produced their failures are detected by processes P_2 and P_4 after their event indices $EI_2(2) = 9$ and $EI_4(4) = 4$, although the errors had occurred after event indices EREI(2) = 2410 and EREI(4) = 0312. The dashed arrows correspond to the recovery control messages exchanged after detection of the errors.

The sequences of EXECEV tuples recorded on volatile storage before the detection of the error are as follows:

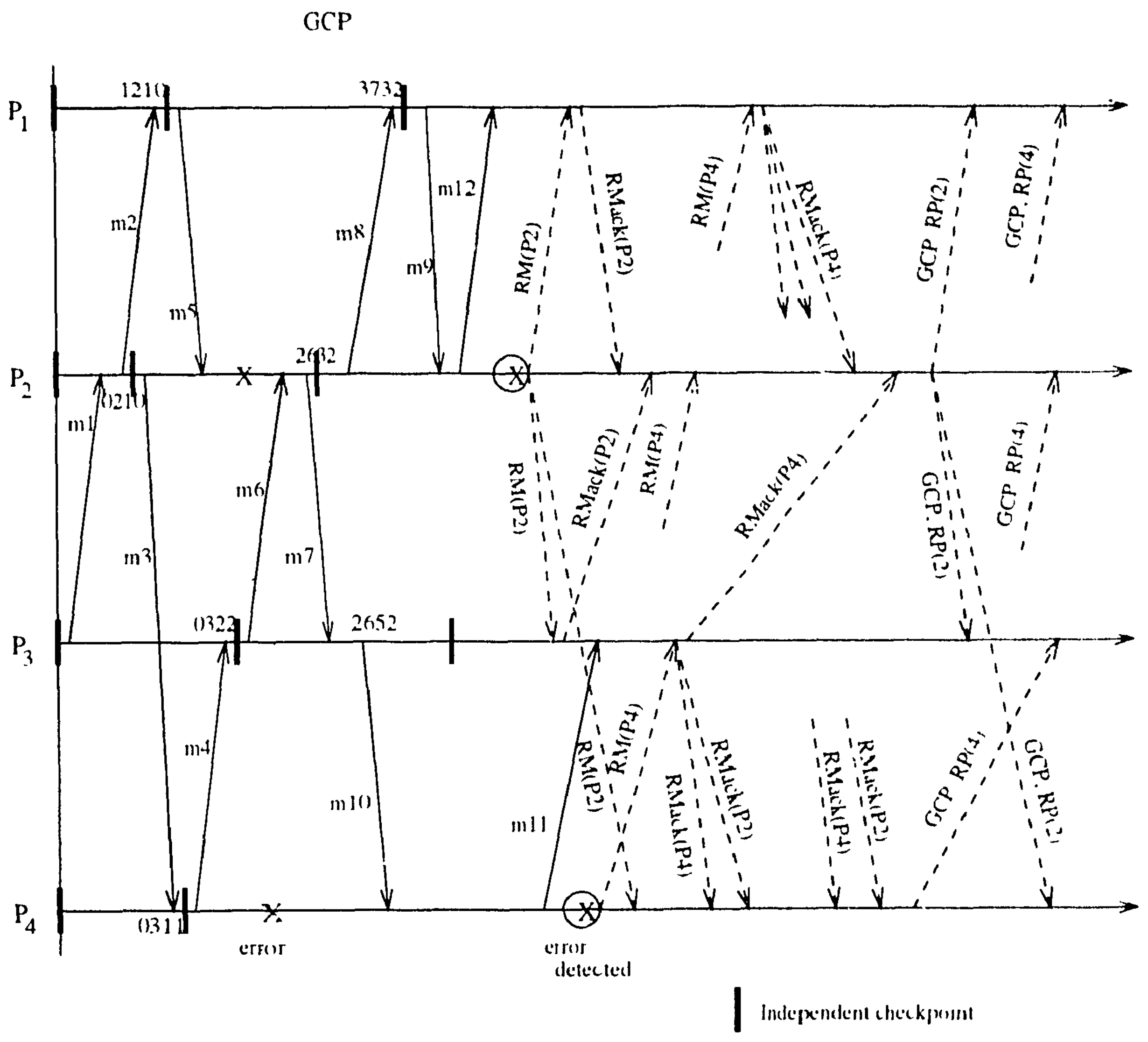


Figure 3.5. An Example to Illustrate Recovery Procedure of Section 3.3.2.3.

For P_1: $(0, P_2, 0210, 1210, m2)$

$(0, P_2, 2210, m5)$

$(0, P_2, 2732, 3732, m8)$

$(0, P_2, 4732, m9)$

$(0, P_2, 4932, 5932, m12)$

For P_2: $(0, P_3, 0010, 0110, m1)$

$(0, P_1, 0210, m2)$

$(0, P_4, 0310, m3)$

$(0, P_1, 2210, 2410, m5)$

$(0, P_3, 0332, 2532, m6)$

$(0, P_3, 2632, m7)$

$(0, P_1, 2732, m8)$

$(0, P_1, 4732, 4832, m9)$

$(0, P_1, 4932, m12)$

For P_3: $(0, P_2, 0010, m1)$

$(0, P_4, 0312, 0322, m4)$

$(0, P_2, 0332, m6)$

$(0, P_2, 2632, 2642, m7)$

$(0, P_4, 2652, m10)$

$(0, P_4, 2654, 2664, m11)$

For P_4: $(0, P_2, 0310, 0311, m3)$

$(0, P_3, 0312, m4)$

$(0, P_3, 2652, 2653, m10)$

$(0, P_3, 2654, m11)$

The MREI tuples saved in stable storage (as a part of the independent checkpoints) corresponding to the indicated places are:

For P_1: (0000), (1210), (3732)

For P_2: (0000), (0210), (2632)

For P_3: (0000), (0322), (2652)

For P_4: (0000), (0311)

Because both P_2 and P_4 detected the error, both may initiate the recovery procedure. P_2 executes the following:

1. It finds the *ICP* (2) (= 0210) that has its $EREI_2(2) \leq 4$ and discards the ICP with the (2632) tuple from stable storage.
2. It increments its *IN#* to 1.
3. * It broadcasts the recovery message $RM(P_2, 1, 0210, 2410)$ to all other processes and waits for their responses.

   * It receives $RMack\ (P_2, 1, 2210)$ and $RMack\ (P_2, 1, 0332)$ from P_1 and P_3 respectively, but it receives a recovery-message $RM(P_4, 1, 0311, 0312)$ from P_4. Therefore it finds *ICP* (2) again, this time with $EREI_2(2) \leq 3$. This is $ICP(2) = 0210$ and is the same as before. It then receives $RMack\ (P_4, 1, 2210)$ and $RMack\ (P_4, 1, 0332)$ from P_1 and P_3
4. * The restart point $RP(2) = 0210$.

   * The global consistent checkpoint tuple $GCP = (2432)$ is determined.
5. The *GCP* is saved in stable storage.
6. It sends the $RP(2)$ and *GCP* to all other processes.
7. It receives $RP(4) = (0000)$ and $GCP = (2432)$ from the other failed process P_4.
8. It restarts from its $RP_2(2) = 2$.

   * continue execution up to the event free from the effects of both the error states in the system, resending message m3 to P_4 and re-receiving message m5 resent by P_1.

* (the local state S4, *MREI*(2) = 2410, *IN#* = 1) is saved in stable storage.

* message m6 is saved in stable storage as part of the global checkpoint.

9. The stable storage is cleared of all the other *ICP* s except the *GCP*.

Process P_4 executes similar steps and saves its local state S2 corresponding to *MREI*(4) = 0312 in stable storage.

When the non-failed process P_3 receives the first recovery-message *RM*(P_2,1,0210,2410) from P_2, it executes the following:

1. It increments its *IN#* to 1.
2. It finds its *ICP* (3) = 0322. It discards its checkpoint after this ICP.
3. It finds in its volatile storage the EXECEV tuple with *MREI*(3) = 0332 and sends *RMack* (P_2,1,0332) to P_2.
4. Before receiving the *GCP* and *RP*(2) from P_2, it stores the received message m11 in a buffer. It also receives *RM*(P_4,1,0311,0312) from P_4 and finds again its *ICP*(3) = 0322 and *MREI*(3) = 0332. It sends its *RMack*(P_2,1,0332) to P_4. It also sends *RMack* (P_4,1,0332) to P_4 and P_2.
5. It receives *GCP* and *RP*(2) from P_2 and *GCP* and *RP*(4) from P_4.
6. It restarts the execution from its *RP*(3) = 0322.
7. It resends message m6 with *IN#* = 1 to P_2.
8. It saves the local state S3 corresponding to *MREI*(3) = 0332 and *IN#* = 1 in stable storage.
9. Other events after (0332) are discarded from the volatile storage. The message m11 in buffer is also discarded.
10. All *ICP* s before *GCP* are removed from the stable storage. All EXECEV tuples before *GCP* are removed from volatile storage.

When the non-failed process P_1 receives the first recovery-message $RM(P_2, 1, 0210, 2410)$ from P_2, it executes similar steps and saves its local state S2 corresponding to MREI(1) = 2210 in stable storage.

The global consistent checkpoint obtained is maximal. Also, twenty recovery control messages are needed to execute the recovery procedure and to obtain the GCP. If the underlying communication subsystem is fully connected, the recovery takes three time units. Eight time units are needed to terminate the recovery procedure if the physical communications connectivity is as shown in the system interaction diagram of Figure 2.2. A larger number of time units is required because of the large distance between processes P_4 and P_1.

3.4. Summary

In this Chapter we have proposed certain algorithmic procedures for checkpointing/rollback recovery in distributed systems based both on the pre-planned and the un-planned approaches.

In the pre-planned approach, error recovery is based on non-fail-stop processes coordinating their checkpointing actions and system restarting from process states obtained from these checkpoints. For this purpose, we have presented two efficient checkpointing procedures and an optimal rollback recovery algorithm with its fault-tolerant version. These checkpointing procedures are based on our maximally reachable event index tuple, which ensures that the checkpoints we obtain are the most recent and consistent possible system states for the processes initiating the procedure. Thus the domino effect is avoided. In addition, our checkpointing algorithms do not interfere with the progress of the underlying system computations. Our procedures can be invoked by any process in the distributed system, periodically or upon failure or can be initiated concurrently and independently by more than one process. Furthermore, their successful termination is guaranteed in spite of the occurrence of any process failure during their invocation. Our optimal backward recovery requires a minimal number of processes to roll back and provides us with a maximum global consistent state after failures. Livelock problems associated with rollback recovery are avoided, since the consistency of message exchanges is retained when a rollback to a consistent checkpoint is executed. These procedures thus meet most of the requirements for ideal checkpointing and recovery procedures except that stable storage is needed to record the local state along with other information after every transition by each process in the system and the control messages of $O(N^2)$ are first required for checkpointing and additional control messages of $O(N^2)$ are again required for rollback.

In the un-planned approach, the problem of fault-tolerance in distributed systems is addressed by providing efficient procedures for recovery in such systems based on the stable storage and volatile storage requirement by the processes. Our recovery procedures do not require the application of an intrusive checkpointing procedure, but use contextual information exchanged between the processes during normal system progress.

In the first approach the processes are assumed to be fail-stop and are not required to take any checkpoints except the initial state. The processes only need to log the message sent or received along with other information in stable storage. During recovery the processes need to replay initially from their initial state and later from their recovery points up to their subsequent recovery points to obtain the local states corresponding to these recovery points. Each process then considers the information recorded after its established recorded point and a minimal number of computations are redone. Also, only the affected processes are required to perform minimal rollback. The procedure described is for single process detecting error and no other failure in the system is allowed during recovery. The control messages required are

Σ#(out-neighbor(P_i)), for every process P_i, which amounts to the $O(N^2)$ in the worst case if each process has every other process as its out-neighbor.

In the second approach for the un-planned strategy, the processes are assumed to be non-fail-stop and at irregular intervals each process independently saves its independent checkpoint in stable storage. The message log is saved in volatile storage thus making the stable storage requirement to be minimal. Thus during recovery the processes need to replay only from their last correct independent checkpoint and not from their initial state. The recovery procedure provides us with a maximum global consistent state. Also, only those processes are required to restart that are affected by error propagation and only those application messages are retransmitted that are affected by error. Our recovery procedure can be used to recover from any number of process failures in the system, including a total failure of all processes. $O(N)$ control messages are required for rollback and obtaining a global consistent state in a system of n processes with single process failure. Therefore, a recovery procedure using this approach meets all the requirements for an ideal recovery procedure. Moreover, proofs of correctness are provided for these procedures and absence of orphan messages, message losses, and duplications is shown.

The next Chapter will concentrate on fault-tolerance in communication protocols and recovery procedures for fault-tolerance based on the un-planned approach will be presented.

CHAPTER 4

FAULT-TOLERANCE IN COMMUNICATION PROTOCOLS

4.1. Introduction

Protocols that su ·.t modern communication systems are becoming more complex and sophisticat.... High reliability and performance in the presence of errors such as unexpected message reception are required for a number of safety-critical, real-time and distributed applications of such protocols. Protocols are not perfectly designed, and some errors may occur during operation. Classical research on protocol design concentrated only on satisfying the well-known safety and liveness properties, to verify the absence of syntactic and semantic design errors. Recently, the design of fault-tolerant protocols has been addressed, and formal protocol specification models are being modified to accomodate the specifications of fault-tolerant requirements [KKMS92, Mal90, SAAA95].

This Chapter addresses the problem of fault tolerance in computer communication protocols, modelled by communicating finite st·te machines, by providing an efficient algorithmic procedure for recovery in such systems. Even when the communication network is reliable and maintains the order of messages, any kind of transient errors that may not be detected immediately could contaminate the system resulting in protocol failure. To achieve fault-tolerance, the protocol must be able to detect the error, and then it must recover from that error and eventually reach a legal (or consistent) state, and resume its normal execution. A protocol that possesses the latter feature of recovering and continuing its execution starting from a legal state is also called a self-stabilizing protocol. The issue related to recovery is tackled here and efficient procedures for the recovery in communication protocols are described.

The recovery procedures make use of the contextual information exchanged between the processes during normal system progress.

4.2. A Formal Model for Communication Protocols

The use of a collection of Communicating Finite State Machines (CFSMs) as a natural and intuitive modeling formalism for communication protocols has been motivated by the observation that protocols can be characterised by event-driven processes that communicate with each other by exchanging messages through uni-directional First-In-First-Out (FIFO) channels [Boch78].

Definitions

A CFSM in a system of N CFSMs, each specifying a process in a protocol, can be formally defined by the quadruple $CFSM_i = (S_i, s_{0_i}, M_{ij}, T)$, where:

S_i is the set of *internal states* of process P_i,

$s_{0_i} \in S_i$ is the *initial state* of P_i,

M_{ij} is the set of messages sent by P_i to other processes (MS_i) and messages received by P_i from other processes (MR_i),

T is a partial transition function: $S_i \times M_{ij} \rightarrow S_i$

(we can say that a message m belonging to an M_{ij} is a label for a transition).

Figure 4.1 shows a protocol example consisting of six CFSMs specifying a part of the ISDN user part of Common Signalling System Seven (CSS 7). This example will be used throughout the rest of this Chapter, and it was also considered by Kakuda [Kak91]. In the figure, the label of a transition consists of: 1) the type of transition: a -(+) sign prefixing the label denotes a sending (receiving) transition, 2) the name of the message, and 3) the name of the process to (from) which the message is sent (received) in the case of a sending (receiving) transition.

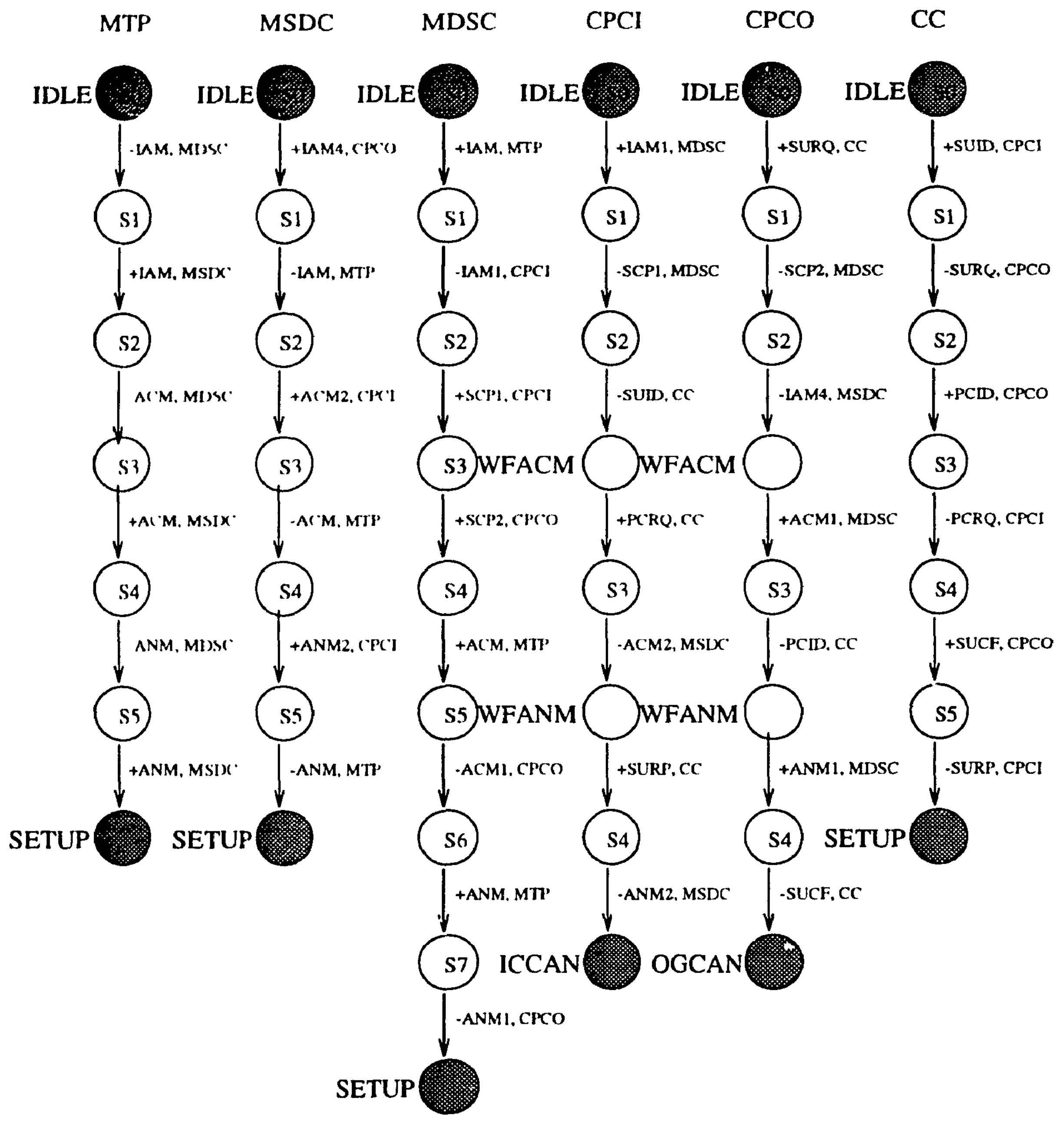


Figure 4.1. Protocol Example: part of the ISDN user part of CCITT's CCS 7

A transition at a state s of process P_i for message m is said to be specified if $T_i(s,m)$ is defined in the CFSM that models P_i. A global state of a protocol is a pair $<S,C>$, where $S = (s_1, s_2, ..., s_n)$ and $s_1, s_2, \cdots, s_n$ represent current states of processes $P_1, P_2, \cdots, P_n$, respectively, and $C = (c_{ij}$, for all $i \neq j$, and $1 \leq i, j \leq N)$ represents the current states of the channels. A global state $<S,C>$ is said to be *reachable* from the initial global state $<S_0, C_0>$, denoted by $<S_0, C_0> \overset{*}{=} <S, C>$ iff $<S_0, C_0> = <S_1, C_1> \overset{*}{=} <S, C>$; that is, there exists an execution path consisting of an interleaving of message receptions and transmissions that takes the protocol from the initial global state $<S_0, C_0>$ to $<S, C>$. A global state $<S, C>$ is said to be *legal* (*illegal* or *unsafe*) if $<S_0, C_0> \overset{*}{=} <S, C>$ is (not) true.

4.3. Issues Related to Fault-Tolerant Protocol Design

There are two types of errors that should be considered for the design of fault tolerant communication protocols: 1) protocol design errors, and 2) operational errors. Because of the increasing importance of computer communication based applications and their criticality, the design of reliable computer communication protocols plays an important role in providing effective and responsive communication services. Current research directions in communication protocols require that a correct protocol design must ensure three types of properties: liveness, safety and responsiveness [SAAA95].

Much research has been done for the development of formal methods for the design of communication protocols. These methods tend to follow one of two design approaches, namely the analytic or the synthetic approaches [PrSa91]. Independent of the approach used, two types of properties must be guaranteed in any protocol design, the safety properties and the liveness properties. The safety properties of the protocol ensure that the protocol never enters an undesirable state. These properties include freedom from deadlocks and absence of unspecified reception error. The liveness properties of the protocol ensure that the protocol performs its intended functions with respect to the service specifications.

Using a specification language or model, the protocol is designed so that the specified protocol service will be provided without encountering either of two types of protocol design errors: i) syntactic errors, which cause the protocol to deadlock, and therefore affect the safety of the protocol due to the presence of an unspecified reception or a deadlock state, and ii) semantic errors, which cause the provision of an incorrect service to the distributed protocol users and therefore affect the liveness of the protocol.

The third type of property for a correct protocol design, responsiveness, possesses the following two features: 1) timeliness, which respects the timing requirements of the protocol specification, and 2) fault-tolerance or stabilization, which recovers the system to a legal state from an illegal state should a protocol error be generated during the operation of the communication software. In this thesis we only address the design of fault-tolerant communication protocols.

The operational errors are related to the environment in which the implementation is executing. These errors, often referred to as transient errors, may change the state of a system, but not its behaviour [Schn93]. We assume that the abstract state of a system may be corrupted, but the system itself is inviolable (its behaviour remains intact). Transient failures may change the global state in a system by corrupting the local state of a process as represented by memory or program counter or by corrupting message channels. All these conditions lead to some form of synchronization loss. The property of fault-tolerance models the ability of a system to recover from transient failures under the assumption that they do not continue to occur. Given that we can never eliminate transient failures, a fault-tolerant system meets a stronger notion of correctness. That is, should a transient failure occur, resulting in an inconsistent system state, then regardless of the failure's origin, the system will eventually correct itself to a consistent system state.

By way of example, coordination loss is examined within distributed systems [Schn93]. This example is adapted from Gouda and Multari [GoMu91] and Multari [Mult89]. "Informally, coordination is said to be lost at a given global state of a distributed program if and only if the local states of the different processes in the program, though each of them may be correct in its own right, are inconsistent with one another in the given global state." This phenomenon has numerous causes, many of which are indistinguishable once such an event has occured. These include:

1) *Inconsistent initialization*: The different processes in the system may be initialized to local states that are inconsistent with one another.

2) *Transmission errors*: The loss, corruption, or reordering of messages may result in an inconsistency between the states of sender and receiver.

3) *Process failure and recovery*: If a process returns to service after "going down", its local state may be inconsistent with the rest of the processes.

4) *Memory crash*: The local memory of a process may crash, causing its local state to be inconsistent with the rest of the processes.

Traditionally, each of these issues has been handled separately, one at a time, and separately for different protocols.

4.4. Related Work in Fault-Tolerance for Recoverable Protocols

There has been a considerable effort in the research community for the development of checkpointing and rollback recovery algorithms for fault-tolerance application. Most of these algorithms have been designed specifically for distributed databases and distributed programs. However, we would like to use checkpointing and rollback recovery in the context of communication protocols. Recently Gambhir [Gamb92, GaFr91] has presented an algorithm for detection and isolation of faults in software providing network services. However, they do not provide any procedures for recovery. Kakuda has introduced checkpointing and recovery procedures to be used for communication protocols. They have characterized fault-tolerant protocols as follows [KKMS92]:

> "A correct state is a global state reachable from an initial global state through a sequence of state transitions such that each state transition is not triggered by any error event. It is a sequence of global states that is passed through by the set of processes before entering an incorrect state. A sequence of state transitions from an initial global state to a correct state is said to be a correct sequence of state transitions. Incorrect states are defined as global states that are not correct states. A sequence of state transitions, such that the first state transition is triggered by an error event and all intermediate global states except the final global state are incorrect states, is called an incorrect sequence of state transitions. A protocol is said to be fault-tolerant if, for each incorrect sequence of state transitions in the protocol, the final global state in the sequence is a correct state".

Kakuda's algorithms are based on the concepts of maximally reachable state and maximally executable sequence, where another process P_j must reach a maximally reachable state L^{ij} for process P_i by executing state transitions given in the maximally executable sequence R^{ij} of process P_j for process P_i. In order for other processes to determine the maximally reachable state and maximally executable sequence a process has to add special information of Rs and Ls for all other processes and its own transition sequence and state reached, to the message usually exchanged among processes. This results in a large overhead to every normal message sent.

Using this added information, the message exchange sequence executed by other processes is always memorized. As a result a large storage requirement is necessary for information of the maximally executable sequence of every other process and the maximally reachable state of every other process. In practice, the maximally executable sequence that starts from the latest saved recovery points in all processes is all that needs to be stored.

In Kakuda's proposed scheme [KKMS91], recovery points in a process that initiates checkpointing are predetermined based on the maximally reachable state and maximally executable sequence. The algorithm performs rollback by restarting from a saved checkpoint and forgetting transmitted messages in the sequence of executed state transitions from the saved checkpoints and also forgetting receiving those messages if they had already been received or discarding them upon their reception. By discarding state transitions from the saved checkpoints, even if they are free from the effects of error, Kakuda's algorithm does not offer minimal rollback. Kakuda has presented the recovery algorithm for the case of a single process with state error considering the reception of an unspecified message as error event and has not considered multiple processes in error.

Gambhir & Frisch [GaFr91] presented an algorithm for the detection and isolation of faults in software providing services. The algorithm combines the results of software static analysis with an event-driven monitoring algorithm. Static analysis is used to generate a structure that describes all possible execution sequences of the network services software. The monitoring algorithm uses this structure to track the progress of the processes and upon fault detection uses the structure to identify the process states at which the fault occurred. Their approach provides the two sets of LAST attributes for each process, which give the boundaries between which the fault should have occurred where one of the boundaries is a consistent state free from the effects of error. The concept of LAST attributes is similar to the concept of maximally reachable sta'e as given by Kakuda where both the LAST attributes and the maximally reachable state identify the states at which other processes must have been. In Kakuda's approach maximally reachable states are obtained dynamically whereas in Gambhir's approach the LAST attributes are predetermined and stored and their information can be obtained by the machine that runs the monitoring system and has the input-output of the formal specification stored in its attached database. Gambhir *et al* do not provide any comments or procedures for checkpointing and rollback recovery. The LAST attributes can be taken as consistent states to which the system may roll back.

Detailed analysis of these algorithms can be found in [Agar94].

4.5. Procedures for Fault-Tolerance Based on Un-Planned Approach

In this Section, we present our recovery procedures, which are based on the unplanned approach of logging all the message transitions and restarting after failures from the obtained restart points. Two approaches are considered for the storage requirement needed to implement our recovery procedures.

4.5.1. Stable Storage Requirement for Message Logging

The executed event tuple (EXECEV tuple) for the sender process P_i consists of the destination process P_j, the type of event (i.e., transmission), $MREI(i)$, and the message m. This record will be denoted by $(P_j, -, MREI(i), m)$ and is stored with the process' execution history on stable storage. Similarly, the EXECEV tuple for the receiver process P_j consists of the source process P_i, the type of event (i.e., reception), $MREI(j)$, and the message m, denoted by $(P_i, +, MREI(j), m)$ and is also stored on stable storage.

In the following, we provide an informal description of our recovery procedure. The in-neighbor (out-neighbor) set of each process is defined to be the set of processes from (to) which it can receive (send) messages according to the protocol specifications.

Informal description

Our recovery procedure does not require any intrusive periodic checkpointing procedure. In our approach, we use two phases. First, after detecting an erroneous message, the processes involved in the protocol will move to a blocking state at which no transmissions can occur. At this state, each process will use the recorded contextual information to determine the effects of the error on itself and will decide whether a rollback is needed or not. Each process determines its recovery point with respect to the message that caused the error and then propagates its recovery point to its out-neighbors. After each process receives the recovery point from each of its in-neighbors, the second phase starts. In this phase, the process moves to its restart state, which depends on its recovery point, and traces the recorded history. Only contaminated messages will be resent and receptions of non-resent (non-contaminated) messages will be replayed from stable storage. Three possibilities exist when considering each of the events recorded after the recovery point: 1) reception event to be replayed if the transmitted message is not erroneous, 2) reception event to be waited for if the transmitted message is erroneous, and 3) transmission event to be resent again if it occurred after the recovery point. Only after considering all the recorded events, the recovery procedure terminates and the process will resume its normal execution. Our recovery procedure terminates and is free from design errors. To prove this, we will show that every retransmitted message will be waited for and every waited for message will be retransmitted and finally every replayed message is not going to be retransmitted. These features amount to freedom from deadlocks and unspecified receptions during recovery. In our procedure, we assume that a process is able to detect the occurrence of a fault caused by an erroneous message reception.

Recovery Procedure

When process P_j detects an error caused by the reception of a message emanating from process P_i, it initiates the recovery procedure following the steps of Section 3.3.1.1(A) for recovery in distributed systems. Similarly, other processes follow the steps of Section 3.3.1.1(B) for recovery.

Figure 4.2 shows a transition diagram illustrating our recovery procedure. A blocking state corresponds to any protocol state at which no transmission of protocol messages will occur. A restart state in a process could be any state in that process determined by the recovery procedure and corresponds to the recovery point for that process.

4.5.1.1. Correctness and Complexity

The correctness of our procedure can be proven exactly on the similar lines of Section 3.3.1.3 by showing that its two constituent phases will terminate properly. In the first phase, we can show that during the tracing of the recorded history, non-progress problems, such as deadlocks and unspecified receptions, will not occur. In particular, we can show that for every retransmitted message there will be a process waiting for that message, and for every replayed message by one process, there will be no corresponding message retransmission by any other process, in the lines similar to Lemmas 3.3, 3.4, 3.5 and Theorem 3.2. In the second phase, we show that eventually each process will receive one recovery control message from each of its in-neighbors, and therefore moves from the blocking state to a restart state. Lemma 3.6

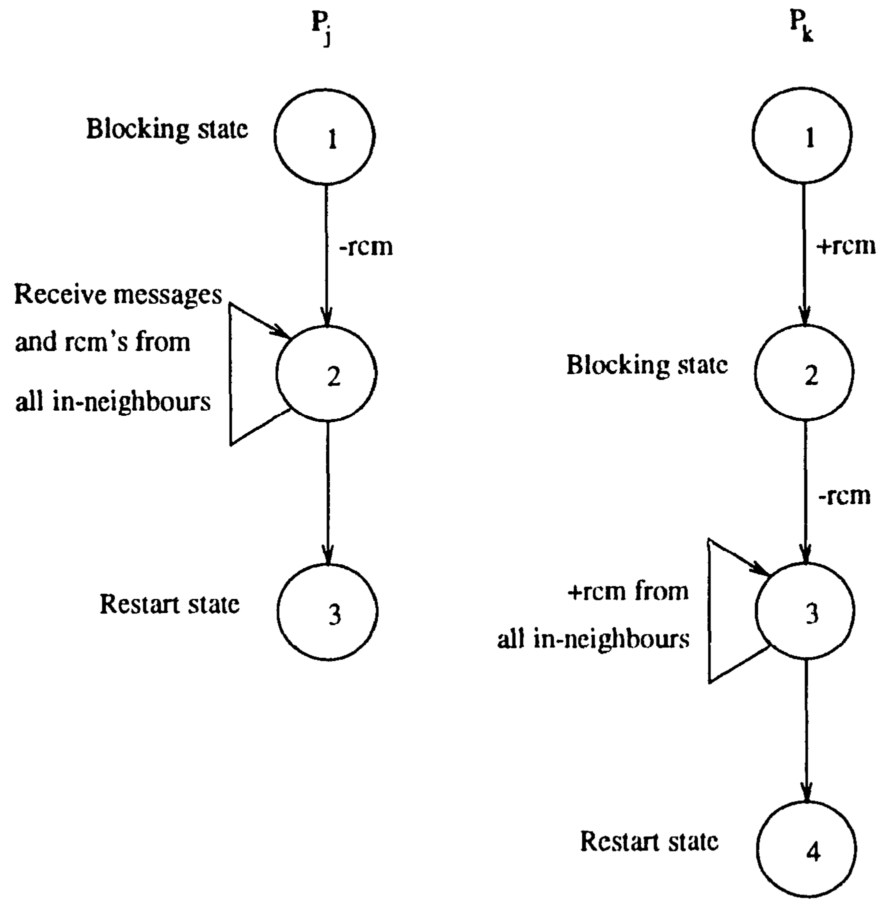


Figure 4.2. Transition Diagram Illustrating the Recovery Procedure of Section 4.5.1

deals with the correctness of this phase and Theorem 3.3 deals with the overall correctness of the recovery procedure.

The complexity of our procedure is also similar to our recovery procedure of Section 3.3.1 and is described in Section 3.3.1.3.

4.5.1.2. Example

In this Section we show an application of our recovery procedure using the example introduced in Section 4.2. The process interaction diagram derived from the protocol specification example of Figure 4.1 is shown in Figure 4.3. The in-neighbor and out-neighbor sets of each of the six processes of the protocol example are shown below:

in-neighbor(MTP) = {MSDC}	out-neighbor(MTP) = {MDSC}
in-neighbor(MSDC) = {CPCO, CPCI}	out-neighbor(MSDC) = {MTP}
in-neighbor(MDSC) = {MTP, CPCI, CPCO}	out-neighbor(MDSC) = {CPCI, CPCO}
in-neighbor(CPCI) = {MDSC, CC}	out-neighbor(CPCI) = {MDSC, CC, MSDC}
in-neighbor(CPCO) = {CC, MDSC}	out-neighbor(CPCO) = {MDSC, MSDC, CC}
in-neighbor(CC) = {CPCO, CPCI}	out-neighbor(CC) = {CPCI, CPCO}

Figure 4.4 shows the time sequence diagram corresponding to our protocol example. Suppose process CPCI detects an erroneous message sent by process CC. The recorded execution histories until an erronoeus message or a recovery control message is received by different processes are as follows:

For CC: (CPCI, +, 103201, SUID);
(CPCO, −, 203201, SURQ);
(CPCO, +, 353623, PCID);
(CPCI, −, 453623, error msg)

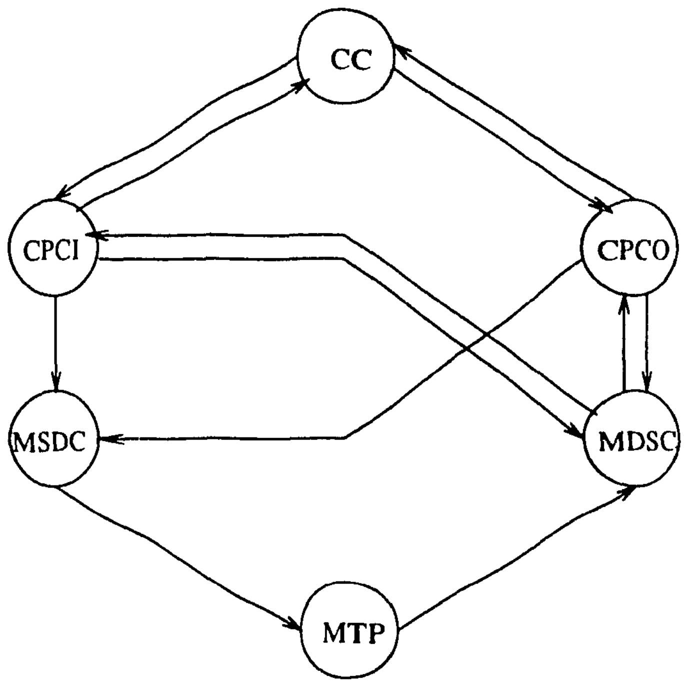


Figure 4.3. Process Interaction Diagram for the Protocol Example

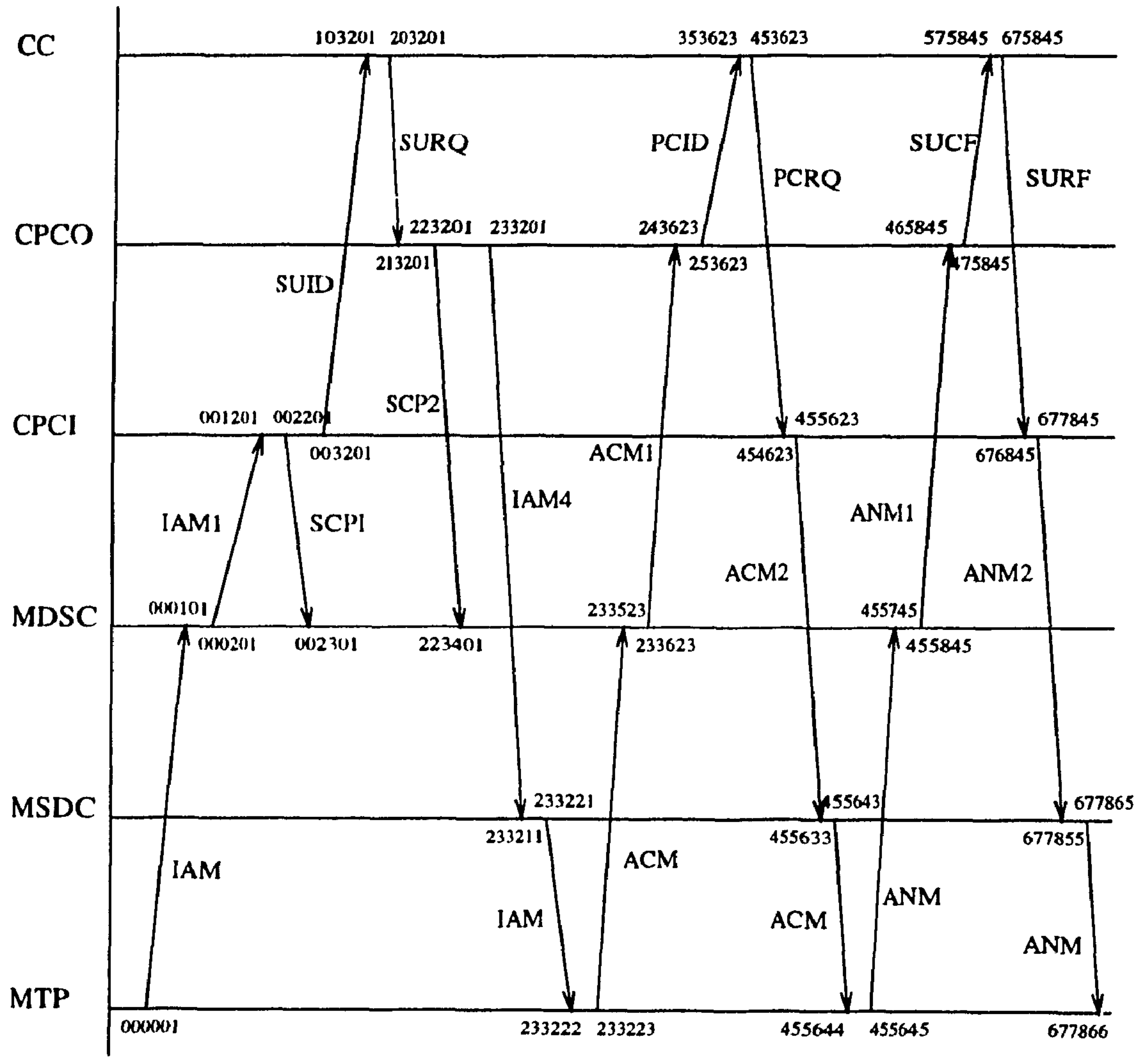


Figure 4.4. Time Sequence Diagram showing Event Index Tuples

For CPCO:	(CC, +, 213201, SURQ);
	(MDSC, –, 223201, SCP2);
	(MSDC, –, 233201, IAM4);
	(MDSC, +, 243623, ACM1);
	(CC, –, 253623, PCID)
For CPCI:	(MDSC, +, 001201, IAM1);
	(MDSC, –, 002201, SCP1);
	(CC, –, 103201, SUID);
	(CC, +, 454623, error msg)
For MDSC:	(MTP, +, 000101, IAM);
	(CPCI, –, 000201, IAM1);
	(CPCI, +, 002301, SCP1);
	(CPCO, +, 223401, SCP2);
	(MTP, +, 233523, ACM);
	(CPCO, –, 233623, ACM1)
For MSDC:	(CPCO, +, 233211, IAM4);
	(MTP, –, 233221, IAM)
For MTP:	(MDSC, –, 000001, IAM);
	(MSDC, +, 233222, IAM);
	(MDSC, –, 233223, ACM)

CPCI initiates the recovery process by sending to all of its out-neighbors including CC (if CC does not belong to its out-neighbor set) a recovery control message that includes its recovery point. Once a recovery control message is received by a process, it computes its recovery point and sends it along its recovery control message to each of its out-neighbors. Figure 4.5 shows the recovery control messages in dashed arrows.

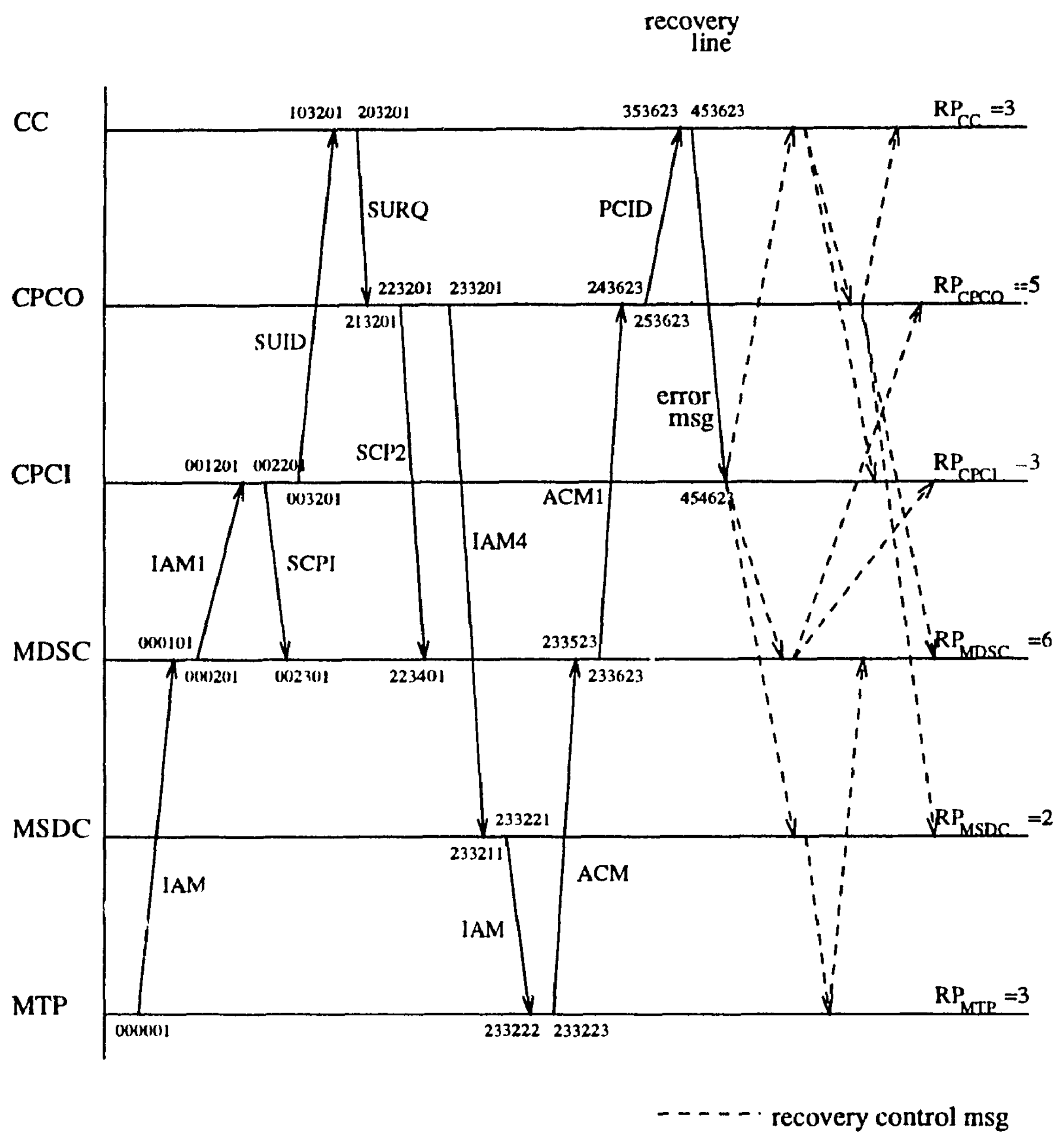


Figure 4.5. Sequence Diagram showing the Recovery Control Messages sent upon failure

Figure 4.6 shows the recovery tree with process CPCI at its root. This tree shows the different control messages exchanged during the application of our recovery procedure. The depth of the tree (which is three) corresponds to the time units required to execute the recovery procedure if the network topology is same as the system interaction diagram of Figure 4.3. The number of edges equal to twelve represents the number of control messages required.

4.5.2. Volatile Storage Requirement for Message Logging

To recover from process errors and failures, and restore the system to a consistent state, a stable log and a volatile log are used. A process P_i saves its local state along with its *MREI*(*i*) tuple and incarnation number (*IN#*), as an independent checkpoint *ICP*(*i*) in stable storage. We assume some mechanism in the system is carried out to checkpoint when the process state is free from errors. The stable storage also stores the initial state of the process as the recovery point and later as obtained by the recovery procedure after failure. In the recovery algorithm, only the *MREI*(*i*) stored in an *ICP*(*i*) is used. Once the restart point has been established, the process will be restarted using the *MREI*(*i*) and the local state. The volatile log is used to save the EXECEV tuples, which consist of the tuples corresponding to the messages received and messages transmitted as discussed in Section 3.3.2. For example, for Figure 4.4, when process CC receives message SUID, it saves (0, CPCI, 003201, 103201, SUID) on volatile storage. Similarly, when CC sends SURQ to process CPCO, it saves (0, CPCO, 203201, SURQ) on volatile storage and so on. No messages are required to be logged to the stable storage during execution.

4.5.2.1. Proposed Recovery Procedure

In this section, we present our recovery procedure, which is based on the

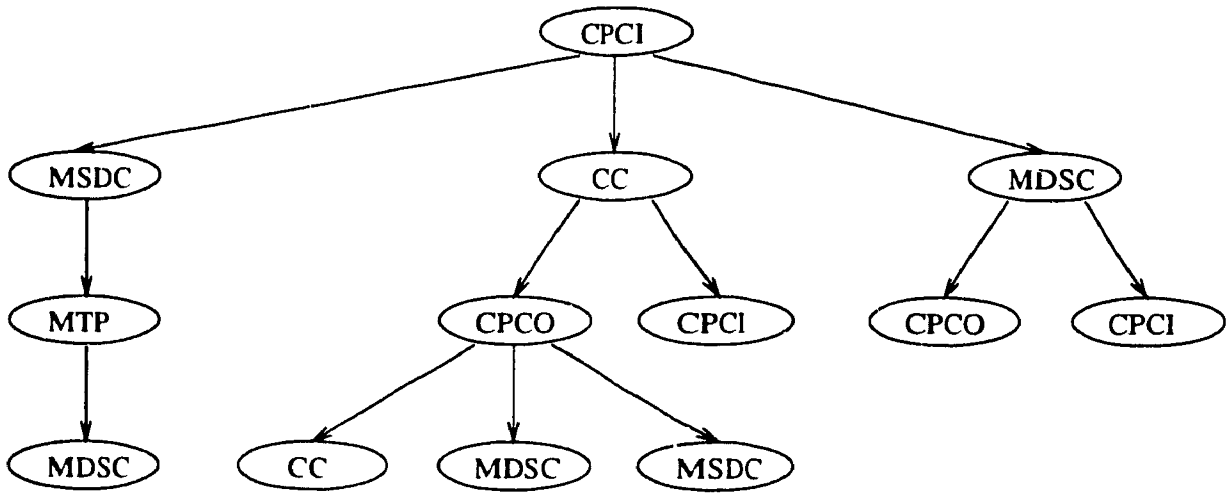


Figure 4.6. Recovery Tree for the Example of Section 4.5.1.2

requirements and concepts introduced in the previous section. We first provide an informal description of our recovery procedure, then we formally describe the steps performed in the procedure. We assume that no failure occurs in non-failed processes during recovery.

Informal description

Our recovery procedure does not require any intrusive pre-planned checkpointing procedure. Instead, independent checkpoints, *ICP*, are saved in stable storage associated with each process P_i. Error recovery is triggered when a process P_j receives an unspecified message m from P_i. $T_j(s_j, m)$ may be undefined because P_j has entered an illegal state, or it may be undefined because P_i has entered an illegal state, thereby sending an incorrect message. P_j starts the recovery algorithm by re-executing from its last independent checkpoint, re-receiving the messages from its volatile storage and finding the mismatch event in comparison to its previous execution. Three cases can be identified, as follows:

casej1: No mismatch event is found. This case occurs when the re-execution proceedes correctly, and the (formerly) unspecified message is no longer found to be unspecified. Thus, the error was in P_j, but the sequence of messages was correct, so the error has not propagated to other processes. Thus P_j simply continues its operation from the point where the original error was detected.

casej2: A message $umsg_j$ previously sent to some process is found to be an invalid message. P_j then determines a recovery point before the sending of $umsg_j$, sends a Recovery Point (RP) message to all other processes, and restarts from its recovery point.

casej3: A message $umsg_m$ is found to be an unspecified message from P_m. P_j then enters the initiator process, where it sends an Initiate Recovery (IR) message to P_m.

When process P_m receives the IR message, it enters into a non-initiator process. It finds its independent checkpoint before the event $umsg_m$ was sent to P_j. If its previous recovery point is greater than this ICP and greater than the event when the $umsg_m$ was sent, then process P_j has wrongfully detected the $umsg_m$. The system enters into a permanent-failure recovery (PFR) algorithm, where it finds a global consistent checkpoint using the independent checkpoints of all processes and the recovery procedure teminates Otherwise, P_m performs re-execution of its recorded history. If P_m has indeed sent $umsg_m$, it finds its recovery point RP_m and sends a Recovery Point (RP) message containing the RP_m to all other processes. The recovery procedure terminates her^e^ for P_m. If instead, P_m now finds some message $umsg_j$ from P_j, which was not previously detected as UMSG sent by P_j, the system enters into the PFR algorithm and the recovery procedure terminates. Another case may be that P_m did not detect any unspecified message during its initial execution but during re-execution it detects $umsg_k$ received from a third process P_k, $k \neq j$. In this case it sends an IR message to P_k, and a $UR(P_k)$ message to P_j indicating that an unspecified message was received from P_k and that P_j may receive an RP_k message from P_k. It then behaves as an initiator process.

Meanwhile, the intiator process P_j receives some control message in response to its IR message to P_m. This control message may be the RP message from P_m, a $UR(P_k)$ message from P_m, an RP message from P_k (if $UR(P_k)$ message received) or the IR message from P_k. The process P_j then either terminates the recovery procedure or enters the PFR algorithm, according to the recovery algorithm.

Finally every process will have its recovery point to restart from.

The Recovery Procedure

For process P_j that receives an unspecified message $umsg_i$ from P_i

When the process P_j detects a loss of synchronization in the system by receiving an unspecified message $umsg_i$ from P_i, it perfoı ms the following execution

Pj_0 increment $IN\#(j)$ by one

Pj_1 find its latest $ICP(j)$. Execute from $ICP_j(j)$ and compare the execution with the recorded execution up to and including the UMSG $umsg_i$ received earlier, and finding the mismatch event.

Case 1: no mismatch event exists

- $IN\#(j) = IN\#(j) - 1$
- exit and continue execution

Case 2: message $umsg_j$ sent to some process is found to be an invalid message

- $RP_j = MREI_j(j)$ before message m_j was sent
- send $RP(P_j, RP_j, IN\#(j))$ to all other processes
- restart

Case 3: some message $umsg_m$ is found to be an UMSG received from process P_m, where P_m is P_i or some other process

- send an Initiate Recovery message $IR(EI(m), EI(j), IN\#(j))$ to P_m to show that an UMSG $umsg_m$ is received. $EI(m)$ is the MREI tuple sent by P_m with the UMSG. $EI(j)$ is MREI tuple for P_j before receiving the UMSG. Recovery point $RP_j = EI_j$ in $EI(j)$

Pj_2 receive messages sent by all other processes into a temporary receiver buffer, until a control message is received from the process P_i or any other process

```
Pj_3 receives control message $CM$
    case a: $CM = IR(EI(j), EI(k), JN\#(k))$, from $P_k$ $(k \neq m)$
        IF $RP_j \geq EI_j(j) > ICP_j(j)$
            - send $RP(P_j, RP_j, JN\#(j))$ to $P_k$
            - send $PFR$ message to all other processes
            - start PFR algorithm
        IF $EI_j(j) > RP_j > ICP_j(j)$
            - send $RP(P_j, RP_j, JN\#(j))$ to all other processes
            - go to Pj_3
    case b: $CM = RP(P_m, RP_m, JN\#(m))$, from $P_m$
        IF $RP_m \geq EI_m(m)$
            - receive PFR message from $P_m$
            - start PFR algorithm
        IF $RP_m < EI_m(m)$
            - IF $RP_m \geq EI_m(j)$
                - restart
            - IF $RP_m < EI_m(j)$
                - find $ICP(j)$ where $ICP_m(j) \leq RP_m$
                - reexecute (without transmitting messages) up to the event
                where $EI_m(j) \leq RP_m$, according to the messages recorded in
                $P_j$'s execution history
                - $RP_j = EI_j(j)$
                - restart
    case c: $CM = PFR$ message from any other process
        - start PFR algorithm
    case d: $CM = UR(P_k)$ msg from $P_m$
        - receive $RP(P_m, RP_m, JN\#(m))$ from $P_m$
```

```
            IF $RP(P_k, RP_k, IN\#(k))$ already received from $P_k$
                - REPEAT_j {
                    - IF $RP_k < EI_k(j)$
                        - find $ICP(j)$ such that $ICP_k(j) \leq RP_k$ AND
                        $ICP_m(j) \leq RP_m$
                        - reexecute (without transmitting messages) up to the
                        event where $EI_k(j) \leq RP_k$ OR $EI_m(j) \leq RP_m$, which-
                        ever is before, according to the messages recorded in
                        $P_j$'s execution history. Then $RP_j = EI_j(j)$
                        - restart
                    - IF $RP_k \geq EI_k(j)$
                        - restart
                } /* end of REPEAT_j */
            ELSE
                - goto Pj_3
    case e:  $CM = RP(P_k, RP_k, IN\#(k))$ from $P_k$
        IF $UR(P_k)$ AND $RP(P_m, RP_m, IN\#(m))$ already received from $P_m$
            - do REPEAT_j
        ELSE
            - go to Pj_3
```

For process P_m, other than P_j

```
case m1:  receive PFR message
              - start PFR algorithm
case m2:  receive $RP(P_j, RP_j, IN\#(j))$ from $P_j$
          IF $(IN\#(m) \geq IN\#(j))$
```

```
            - ignore RP message, exit and continue execution
        IF $(IN\#(m) < IN\#(j))$
            - IF current $MREI_j(m) > RP_j$
                -find $ICP(m)$ where $ICP_j(m) \leq RP_j$
                -reexecute (without transmitting messages) up to the event with
                $EI_j(m) \leq RP_j$
                - $IN\#(m) = IN\#(j)$
                - $RP_m = EI_m(m)$
                - restart
            - IF current $MREI_j(m) \leq RP_j$
                - $IN\#(m) = IN\#(j)$
                - exit and continue execution
case m3: receive $IR(EI(m), EI(j), IN\#(j))$ from $P_j$
        IF $(IN\#(m) \geq IN\#(j))$
            - ignore IR message, exit and continue execution
        IF $(IN\#(m) < IN\#(j))$
            - find $ICP(m)$ where $ICP_m(m) \leq EI_m(j)$
            - IF $RP_m \geq EI_m(m) > ICP_m(m)$
                - send $RP(P_m, RP_m, IN\#(m))$ to $P_j$
                - send PFR message to all other processes
                - start PFR algorithm
            - reexecute from $ICP_m(m)$ up to and including $EI_m(m)$ and compare
            the execution with the recorded execution without transmitting the
            same messages
                case m3a: msg $umsg_m$ is considered as UMSG sent
                    - $IN\#(m) = IN\#(j)$
                    - $RP_m = EI_m$ before $umsg_m$ was send
        - send $RP(P_m, RP_m, IN\#(m))$ to all other processes
        - restart
case m3b: received $umsg_j$ as UMSG from $P_j$
        - $IN\#(m) = IN\#(j)$
        - $RP_m = EI_m$ before receiving $umsg_j$
        - send PFR message to all other processes
        - start PFR algorithm
case m3c: received $umsg_k$ as UMSG from $P_k$
        - $IN\#(m) = IN\#(j)$
        - $RP_m = EI_m$ before receiving $umsg_k$
        - send $IR(EI(k), EI(m), IN\#(m))$ to $P_k$
        - send $UR(P_k)$ to $P_j$
        - send $RP(P_m, RP_m, IN\#(m))$ to all other processes
        - receive_state {
            receive control message $CM$
            m3ca: $CM = RP(P_k, RP_k, IN\#(k))$ from $P_k$
                - IF $RP_k \geq EI_k(k)$
                    - receive PFR message from $P_k$
                    - start PFR algorithm
                - IF $RP_k < EI_k(k)$
                    - IF $RP_k \geq EI_k(m)$
                        - restart
                    - IF $RP_k < EI_k(m)$
                        - find $ICP(m)$ where $ICP_k(m) \leq RP_k$
                        - reexecute upto $EI_k(m) \leq RP_k$
                        - $RP_m = EI_m(m)$
                        - restart
```

```
m3cb: $CM = PFR$ message from $P_k$
    - start PFR algorithm
m3cc: $CM = IR(EI(m), EI(p), JN\#(p))$ where p is k or
any other process
    - IF $RP_m \geq EI_m(m)$
        - send PFR message to all other processes
        - start PFR algorithm
    - IF $RP_m < EI_m(m)$
        - ignore IR message
        - go to receive_state
m3cd: $CM = UR(P_q)$ message from $P_k$
    - receive $RP(P_k, RP_k, JN\#(k))$ from $P_k$
        IF $RP(P_q, RP_q, JN\#(q))$ already received
        from $P_q$
            - REPEAT_m {
                - IF $RP_q < EI_q(m)$
                    - find $ICP(m)$ where
                    $ICP_q(m) \leq RP_q$ AND
                    $ICP_k(m) \leq RP_k$
                    - reexecute (without
                    transmitting messages) up to
                    the event where
                    $EI_q(m) \leq RP_q$ OR
                    $EI_k(m) \leq RP_k$, whichever is
                    before, according to the
                    messages recorded in $P_m$
                    execution history. Then
```

```
                        $RP_m = EI_m(m)$
                        - restart
                      - IF $RP_q \geq EI_q(m)$
                        - restart
                  } /* end of REPEAT_m */
              ELSE
                  - goto receive_state
        m3ce: $CM = RP(P_q, RP_q, JN\#(q))$ from $P_q$
            IF $UR(P_q)$ and $RP(P_q, RP_q, JN\#(q))$ already
            received from $P_q$
                - do REPEAT_m
            ELSE
                - go to receive_state
        } /* end of receive_state */
    m3d: no mismatch event up to $EI_m(m)$
        - continue execution finding the mismatch event
            - IF $umsg_m$ sent, or $umsg_k$ received, or no error
            event
                - find $RP_m$
                - send $RP(P_m, RP_m, JN\#(m))$ to $P_j$
                - send PFR message to all other processes
                - start PFR algorithm
```

4.5.2.2. Correctness

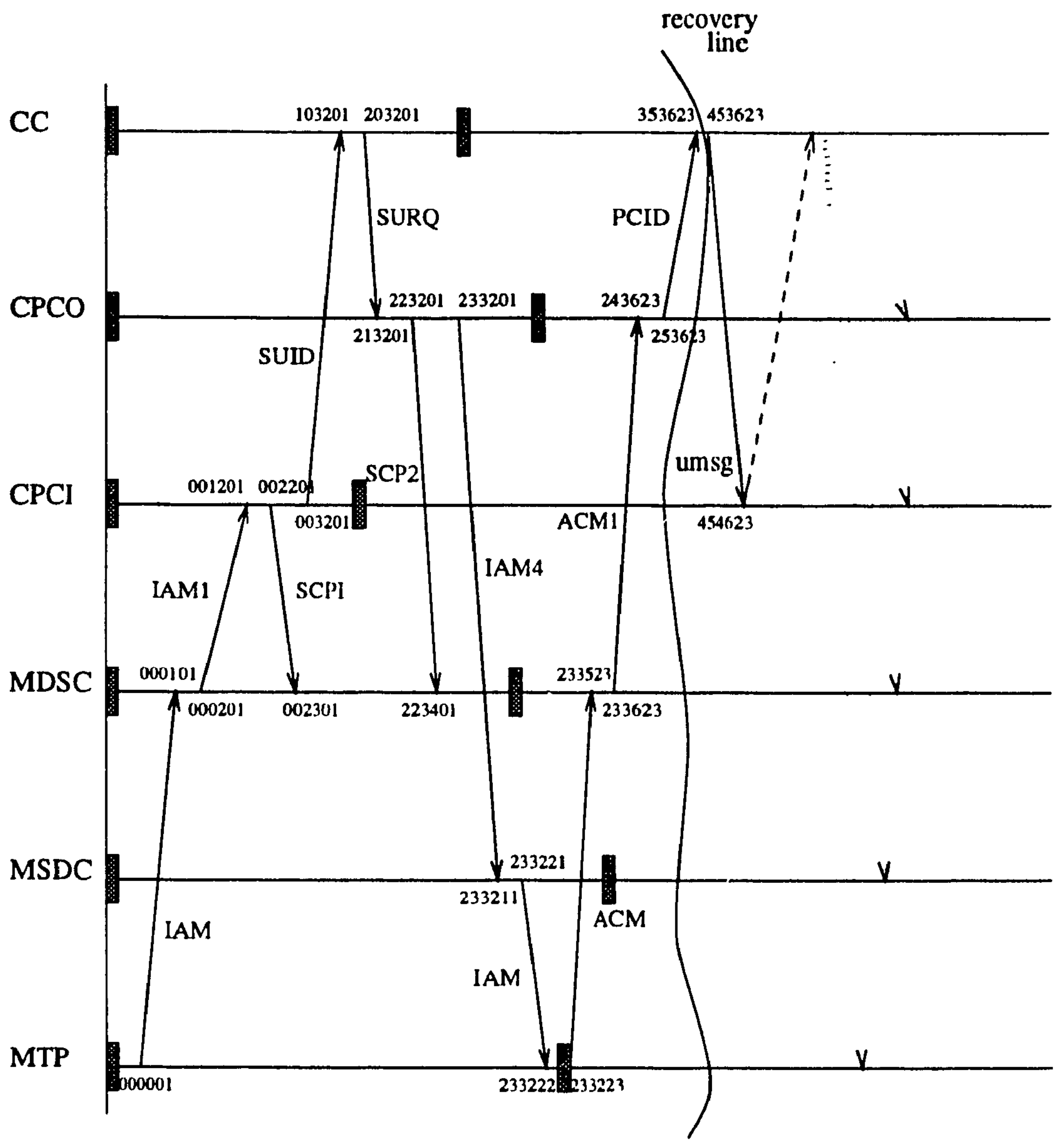
The SPIN package (an automated protocol validation system for Promela models) is used to validate the design. In order to validate a design, we need to be able to specify precisely what it means for a design to be correct. A design can be proven correct only with respect to specific correctness criteria. We are interested in finding the following standard criteria: the absence of deadlocks, logical incompleteness of the protocol specification (e.g., unspecified reception), improper terminations and unexecutable code segments.

We will see in next chapter that our recovery algorithm meets the above criteria by considering an example.

4.5.2.3. Example

To illustrate the recovery procedure, consider the example of Figure 4.4, with the independent checkpoints and recovery messages added when process CPCI receives an unspecified message, *umsg* as shown in Figure 4.7. The sequences of EXECEV tuples recorded on volatile storage before the detection of the error are as follows:

For CC:	(0, CPCI, 003201, 103201, SUID);
	(0, CPCO, 203201, SURQ);
	(0, CPCO, 253623, 353623, PCID);
	(0, CPCI, 453623, umsg)
For CPCO:	(0, CC, 203201, 213201, SURQ);
	(0, MDSC, 223201, SCP2);
	(0, MSDC, 233201, IAM4);
	(0, MDSC, 233623, 243623, ACM1);
	(0, CC, 253623, PCID)
For CPCI:	(0, MDSC, 000201, 001201, IAM1);
	(0, MDSC, 002201, SCP1);



▮ independent checkpoint - - - - - IR(453623,003201,1) msg
RP(CC,3,1) msg

Figure 4.7. Sequence Diagram showing Recovery Control Messages for Section 4.5.2.3

	(0, CC, 003201, 103201, SUID);
	(0, CC, 454623, umsg)
For MDSC:	(0, MTP, 000001, 000101, IAM);
	(0, CPCI, 000201, IAM1);
	(0, CPCI, 002201, 002301, SCPI);
	(0, CPCO, 223201, 223401, SCP2);
	(0, MTP, 233223, 233523, ACM);
	(0, CPCO, 233623, ACM1)
For MSDC:	(0, CPCO, 233201, 233211, IAM4);
	(0, MTP, 233221, IAM)
For MTP:	(0, MDSC, 000001, IAM);
	(0, MSDC, 233221, 233222, IAM);
	(0, MDSC, 233223, ACM)

The event index tuples corresponding to the independent checkpoints saved are as follows:

For CC:	(000000), (203201)
For CPCO:	(000000), (233201)
For CPCI:	(000000), (003201)
For MDSC:	(000000), (223401)
For MSDC:	(000000), (233221)
For MTP:	(000000), (233222)

When process CPCI receives an unspecified message, it increments the incarnation number by one and reexecutes from the local state corresponding to its latest independent checkpoint of (003201). It again finds that *umsg* is indeed the unspecified message. Therefore an *IR(453623, 003201, 1)* message is sent to CC. CPCI makes 3 in (003201) as its recovery point and waits for a control message from any process. After it receives a recovery point message *RP(CC, 3, 1)* from CC, it restarts from its recovery point.

After receiving an *IR(453623, 003201, 1)* message from CPCI, process CC reexecutes from its local state of independent checkpoint of (000000) up to its event tuple of (353623) finding an *umsg* message sent instead of the actual message *PCRQ*. It increments its incarnation number and sends a recovery point message *RP(CC, 3, 1)* to all other processes and restarts.

All other processes, after receiving the recovery point message *RP(CC, 3, 1)* from CC, increment their incarnation number and find their current maximally reachable event index not to be effected by the error event. Therefore, they continue their execution without restarting.

For the similar example considered as in Section 4.5.1.2 using stable storage, we see that six control messages are required in all and it takes 4 time units for the system to recover and to restart from the system state of (353623). In addition, time is required to trace and replay recorded histories at each process, from the initial state (or the recovery points if any previous failures had occurred) in case of recovery procedure described in Section 4.5.1, and from the latest correct independent checkpoints for this Section's recovery procedure. However, since only the erroneous messages are retransmitted, the overall time is minimal.

4.6. Comparison

In this Section we compare our recovery procedure with Kakuda's recovery scheme [Kak91, KKMS92], which deals specifically with recovery in protocol systems. The main differences between our recovery procedure and Kakuda's scheme can be summarized as follows:

1) In our procedure, no periodic checkpointing is required to establish a recovery line to be used later for recovery. However, in Kakuda's scheme, periodic checkpointing is needed since processes roll back to their latest checkpoint and restart from there. This implies that in our procedure, there is no overhead associated with checkpointing, whereas in Kakuda's scheme checkpointing is intrusive and is based on a two-phase commit protocol that requires control messages of the $O(N^2)$.

2) Kakuda's rollback-recovery algorithm performs rollback by restarting from a saved checkpoint and forgetting transmitted messages in the sequence of executed state transitions from the saved checkpoints and also forgetting receiving those messages if they had already been received or discarding them upon their reception. By discarding state transitions from the saved checkpoints, even if they are free from the effects of error, Kakuda's algorithm does not offer minimal rollback. In our procedure the recovery line is not the checkpoint line but a consistent state corresponding to the maximally reachable states for the process detecting an error or a consistent state beyond the maximally reachable state. Using our recovery procedure, only the affected processes are required to perform a minimal rollback whereas in Kakuda's scheme all communicating processes roll back to their latest checkpoint.

3) Our procedure requires the exchange of contextual information along with the exchanged message. Kakuda's scheme requires a larger amount of information including the maximally executable sequence to be sent along with the message.

4) Our procedure keeps the amount of message retransmission to a minimum. Only those messages are required to be retransmitted that are contaminated by error events. Kakuda's scheme requires the retransmission of all messages

transmitted by one process after its latest checkpoint.

5) Our recovery procedure requires fewer recovery control messages to be transmitted and also requires less time for recovery (if we suppose that a message from one process to another takes one time unit). For the example considered, the recovery algorithm of Section 4.5.1 requires 12 control messages and 3 time units, the algorithm of Section 4.5.2 requires 6 control messages and 4 time units, and Kakuda's scheme requires 15 control messages and 6 time units. This is because Kakuda's recovery is based on a two-phase commit protocol to coordinate and propagate a recovery line to each of the communicating processes.

6) A large storage requirement is necessary for information of the maximally executable sequence tuples and maximally reachable state of every other process and for periodic checkpoints for Kakuda's approach, whereas in our approach we need to store in volatile storage only the tuples corresponding to the events that were executed. The stable storage is required only for saving the independent checkpoints.

7) Kakuda's approach considers only the occurrence of error events for a single fail-stop process. Our approach allows recovery from multiple processes entering into error states, where errors may be detected immediately or they may have been propagated to other processes.

Consider the example of Figure 4.1, reproduced in Figure 4.8, displaying the concepts of all the approaches considered. The message transitions have been omitted for simplicity. After an error in message *PCRQ* is detected by the receiver process *CPCI*, Kakuda's algorithm rolls back to its checkpoint line, in this case the initial state for all processes. It forgets the transmission and receptions of all the messages. Gambhir's approach detects that there is an error between state $S1$ and $S3$ of process

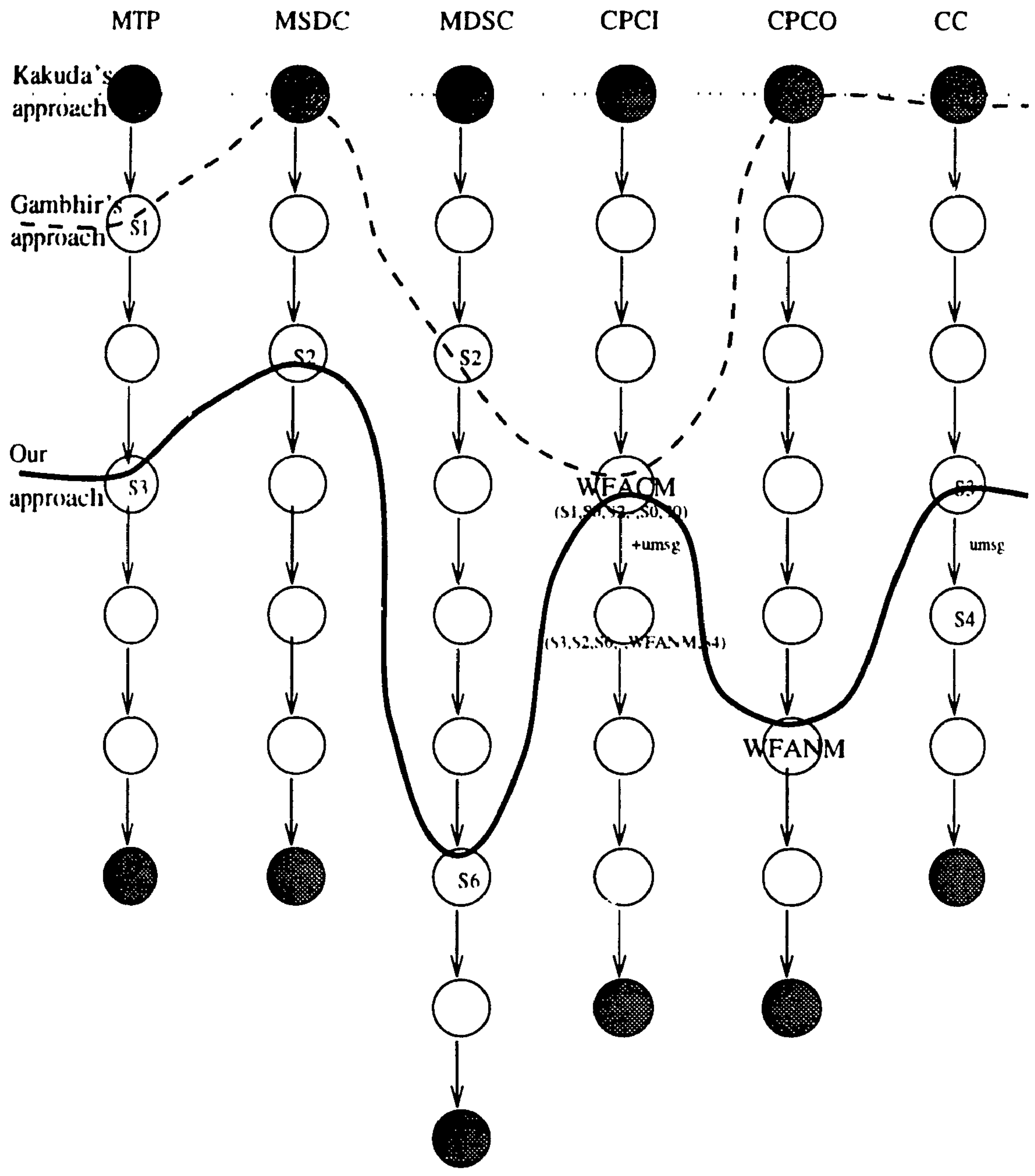


Figure 4.8. Comparison of different approaches

MTP, states $S0$ and $S2$ of process $MSDC$, states $S2$ and $S6$ of process $MDSC$, states $S0$ and $WFANM$ of process $CPCO$ and/or between states $S0$ and $S4$ of process CC, i.e. state $S1$ of process MTP, state $S0$ of process $MSDC$, state $S2$ of process $MDSC$, state $S0$ of process $CPCO$ and state $S0$ of process CC are correct states. Therefore states $(S1,S0,S2,WFACM,S0,S0)$ of the processes MTP, $MSDC$, $MDSC$, $CPCI$, $CPCO$ and CC, in that order, form a consistent state to which the system can roll back. According to our approach the recovery line corresponds to the consistent state $(S3,S2,S6,WFACM,WFANM,S3)$ of the processes in that order. None of the messages except $PCRQ$ are resent or received again since they are not corrupted by the erroneous message *umsg*.

4.7. Summary

This Chapter provides some solutions to the problem of fault-tolerance in computer communications protocols, modelled by communicating finite state machines. We begin by considering the issues related to the fault-tolerant protocol design and focus our attention to the operational (transient) errors that change the global state of the system due to process failures or memory crashes, but not its behaviour.

We have seen in the literature that not much work has been done in the field of applying the checkpointing and rollback-recovery algorithms of distributed systems to communication protocols. Researchers (e.g., GaFr91) have presented algorithms for fault detection and isolation by comparing the input-output sequences of the software implementation with the formal specification stored in its attached database, but not concentrated on fault recovery. Kakuda has introduced checkpointing and recovery procedures to be used for communication protocols where in their scheme fail-stop processes, upon detection of a single error in the system, perform rollback from a predetermined global checkpoint and forget transmitted messages in the sequence of executed state transitions from the saved checkpoints and also forget receiving those messages if they had already been received or discarding them upon their reception.

In this Chapter, we have first introduced an efficient procedure for the recovery in protocol systems for a single error in a fail-stop process. Later, we have presented a generalized recovery procedure that deals with concurrent failures and therefore concurrent initiations of the recovery procedure for non-fail-stop processes. Our procedure uses the contextual information exchanged during the normal progress of the protocol and recorded on volatile storage. On reception of an erroneous message by any process, the receiver process finds by reexecution if its own process state has corrupted or it is indeed a sender process that has failed by sending an unspecified message, therefore not suspecting each time the sender process to be failed. The process involved in sending the erroneous message in the protocol will become aware of the error and will evaluate the effects of that error on its progress. Depending on that effect the sender process will decide on its independent checkpoint from which it reexecutes the recorded history and considers each of the recorded events. Since the errors are assumed to be transient, reexecution from its checkpoint tells if the process itself has an erroneous state or due to some other process in error, it had sent an erroneous message. Also depending on the error event, some of the processes may not be required to roll back, meaning that our procedure forces only the necessary and minimal number of processes to roll back. A message will be retransmitted only if it was corrupted, thus keeping the number of message retransmissions to a minimum.

An example is considered consisting of six CFSMs specifying a part of the ISDN user part of Common Signalling System Seven and is evaluated in terms of recovery control messages and time required to roll back in case of an error message reception. A comparison is made between Kakuda's approach and our recovery procedures and it is found that our approach performs a minimal rollback, requires only affected processes to rollback, requires less information to be exchanged, keeps message retransmissions to be minimum, needs less stable storage, requires fewer recovery control messages to be transmitted, and also requires less time for recovery as compared to Kakuda's approach. Therefore, our generic algorithms are better suited to provide responsive protocols.

CHAPTER 5

DESIGN OF FAULT-TOLERANT PROTOCOLS

5.1. Introduction

Research on stabilization (fault-tolerance) started in the context of distributed computing systems with a classical paper by Dijkstra [Dijk74]. However, the importance of stabilization and responsiveness, for communication protocols as a special type of distributed systems, was only recognized recently.

A fault-tolerant computer system is one that delivers the specified services even in the presence of faults. The first step in fault-tolerant design is to provide modifications to existing protocols such that faults are considered as a part of the system specifications. The design needs to accomodate fault diagnosis and fault recovery times for the specific classes of faults [Mal90].

In the conventional method for design of real protocols, a protocol is manually designed such that all processes in the protocol are controlled by a specified process for recovery from an error event, and different sequences of state transitions are followed for each state where the error event occurs. This report proposes a unified method for design of fault-tolerant protocols for communication systems which incorporates the procedures for fault-tolerance based on our checkpointing and rollback-recovery procedures for protocols as presented in Chapter 4 into the specification and design phases of the communication protocols. Any process where an error event occurs (is detected) becomes an initiator process for recovery from the error event. The method is regarded as a unified approach since it is independent of the inherent functions of any protocols. For any error event that is detected by the reception of an unspecified message, the sequences of state transitions for recovery purposes for the processes required to rollback are generated by our procedures in Chapter 4.

Three different approaches can be clearly identified for the design of fault-tolerant protocols — conventional method, semantic-based, and generic approach. A brief description of each follows.

5.2. Conventional Design Method for Fault-Tolerant Protocols

Design of fault-tolerant protocols involves the following steps [KaKi92]:

1) Specification of protocol structure: A relation for message flow among processes is specified. An example is shown in Figure 5.1 where circles denote processes, arrows represent channels, and labels attached to the arrows denote messages.

2) Specification of protocol behaviour: A relation between transmission and reception of messages among processes is specified. For example, in Figure 5.2, circles denote states, arrows correspond to state transitions, and a minus/plus sign denotes transmission/reception of a message. In other words, the specification of protocol behaviour is equivalent to the specification of sequences of state transitions. This step is divided into the following two substeps:

2.1) Specification of normal sequences of state transitions: transitions of Figure 5.2 as shown in its reachability tree represent normal sequences of state transitions.

2.2) Specification of abnormal sequences of state transitions: reset sequences due to reception of unexpected messages are instances of abnormal sequences of state transitions.

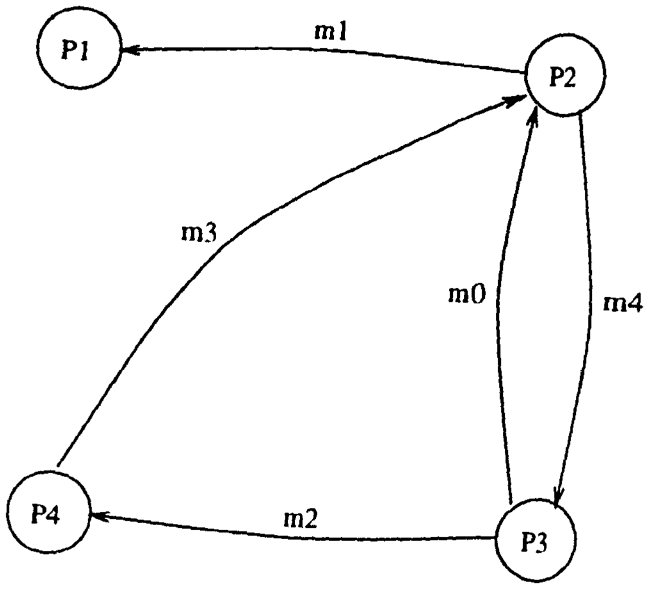


Figure 5.1. Specification of Protocol Structure

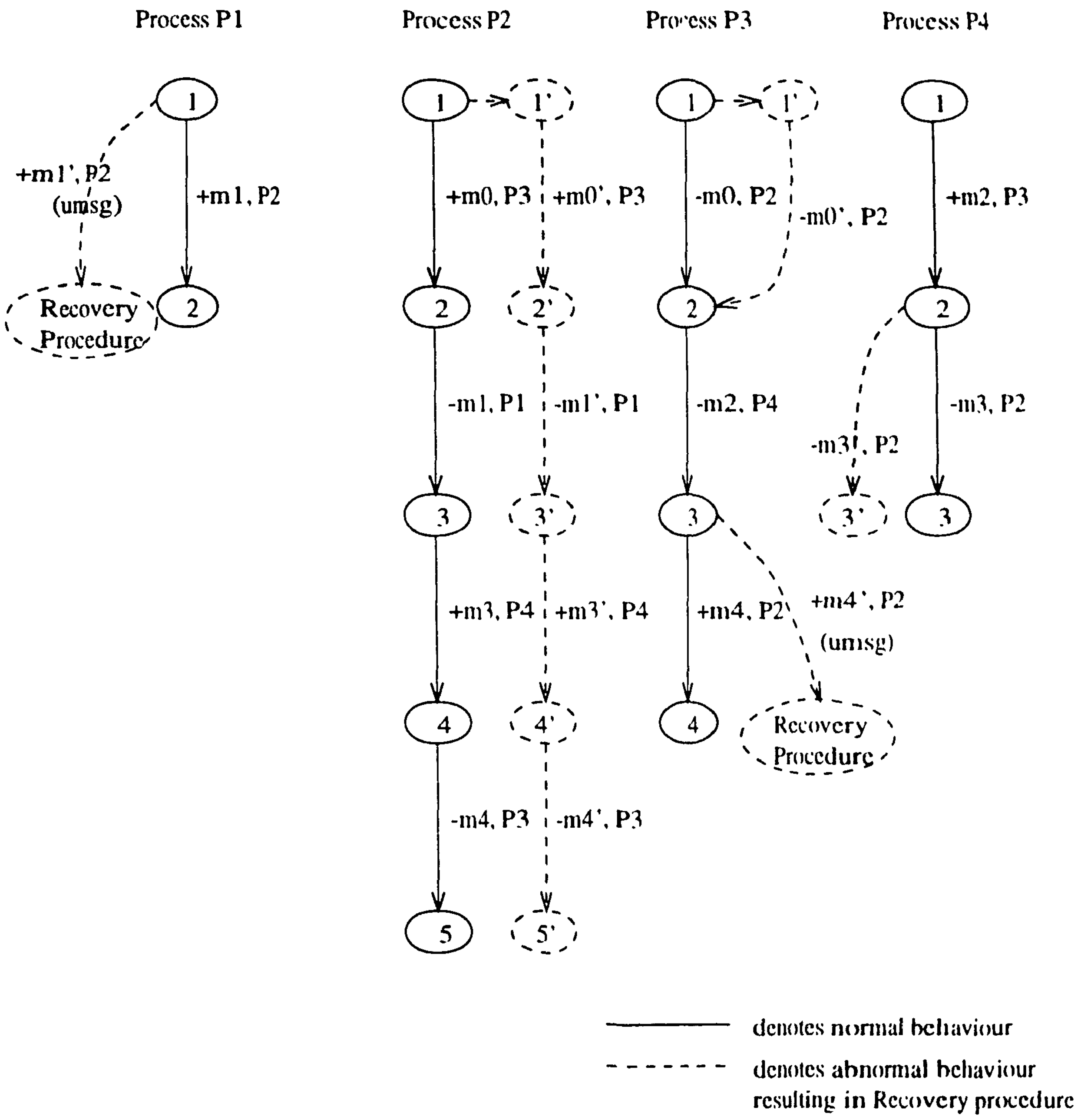


Figure 5.2. Specification of Protocol Behaviour

The number of normal sequences is restricted to those which the protocol designers can perfectly specify, while there are many abnormal sequences because error events can occur at any normal state. It is therefore important to specify abnormal sequences for design of fault-tolerant protocols. Steps 1) and 2.1) are common to the design of any protocol, fault-tolerant or not. Our focus is on step 2.2), which is crucial in the design of fault-tolerant protocols.

When error events occur, real protocols often revert to an initial global state through abnormal sequences, called reset sequences, of state transitions. All state transitions from the initial global state to the reached abnormal state are voided. Also there is a central process that controls transmission and reception of messages in the abnormal sequences, thus the time required for executing all the state transitions in the abnormal sequence is more likely to exceed an assumed deadline.

5.3. Semantic-Based Approach for Fault-Tolerant Protocols

This approach for design of fault-tolerant protocols is to use verification techniques. These techniques aim to prove that the specification meets requirements for fault-tolerant protocols. First, the specifications are flexibly designed without placing any restrictions on them, and then the errors against fault-tolerance are corrected manually.

The first work on verification of fault-tolerant protocols was done by Gouda and Multari [GoMu91]. They propose a mathematical model for verifying self-stabilization of protocols. Kakuda and Kikuno propose an automated method for verifying fault-tolerant protocols [KaKi91]. They model the protocols by an extended finite state machine and apply methods for protocol verification to verify the fault-tolerant properties.

All the design methods for fault-tolerant protocols described using verification method are application-specific. Expertise in protocols is required to correct an erroneous invariant, to encompass all failure modes within the invariant, and to establish time bounds for recovery. Moreover, the modified protocol must be re-validated for syntactic and semantic correctness and therefore, it is a time-consuming approach.

5.4. Generic Approach for Fault-Tolerant Protocols

Another approach for design of fault-tolerant protocols is generic and is based on the structure of the protocol specification, which aims to incorporate procedures for fault-tolerance into the specification and design phases of the communication protocols. In this thesis, we advocate the generic approach. Kakuda *et al* have proposed a method for synthesis of fault-tolerant protocols using checkpointing and rollback-recovery [KKMS92]. Saleh *et al* have also proposed checkpointing and rollback-recovery procedures [SAAA95] to be used in synthesis of fault-tolerant protocols. This method is suitable for design of fault-tolerant protocols, since any process can initiate the rollback-recovery procedure and any illegal sequence in the protocol can revert to an intermediate legal consistent state while retaining consistency in transmission and reception of messages. Only the state transitions from an intermediate consistent state to the abnormal state are voided.

All the design methods for fault-tolerant protocols described using the synthesis method are application-independent. Since they only require the protocol structure and exchange of local information, they are regarded as unified design methods for any protocol and all failure modes, and it is sufficient to prove the correctness / validate the provided algorithms to be able to use them for any fault-tolerant communication protocol regardless of its semantics. However, the procedures and stable storage required for the synthesis are assumed to be fault-tolerant.

In the following Section we specify the sequences of state transitions for recovery purposes using transition diagrams when there is a reception of unspecified message(s) by the process(es) of the system.

5.4.1. Transition Diagrams

Figure 5.3 shows the time sequence diagram for normal behaviour of Figure 5.2. Transition diagrams for the processes involved in the protocol of Figure 5.3 are shown in Figure 5.4, representing the transitions a process has to undergo when it may be an initiator or a non-initiator of the recovery procedure. A process is an initiator when it has received an *umsg* or it may be a non-initiator when an *IR* control message has been received. In the example considered, any of the processes P_1, P_2, P_3 or P_4 may be an initiator process by receiving messages $m1$, $m0$ or $m3$, $m4$, $m2$, from processes belonging to their set of in-neighbors, as error messages $umsg1$, $umsg0$ or $umsg3$, $umsg4$, $umsg2$, respectively. Thereafter, the processes execute the transitions as specified in the transition diagrams of their initiator processes. Processes P_2, P_3 or P_4 may run their non-initiator processes after receiving *IR* messages. P_2 may receive an *IR* message from either P_1 or P_3 since they belong to its set of out-neighbors and the corresponding path of the transition diagram is followed. Similarly, P_3 may receive an *IR* message from P_2 or P_4, and P_4 may receive an *IR* message from P_2. There is no non-initiator process for P_1 since it is not a sender of the application message in the example considered.

Consider the case when process P_1 receives message $m1$ as $umsg1$. It enters into its initiator1 process and sends an $IR(EI[2],EI[1],1)$ message to P_2. Process P_2 may receive this *IR* message after receiving $m3$ or after sending $m4$. It then finds its

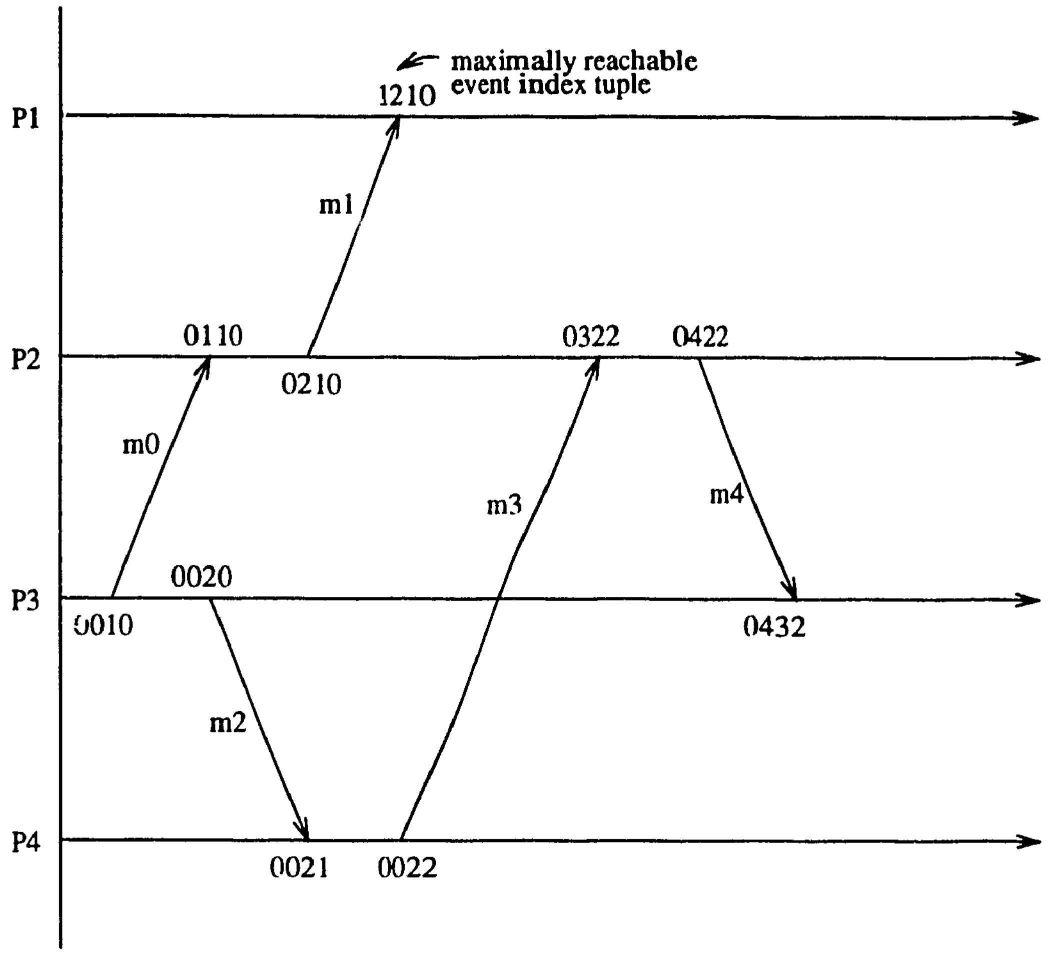


Figure 5.3. Time Sequence Diagram for Figure 5.2.

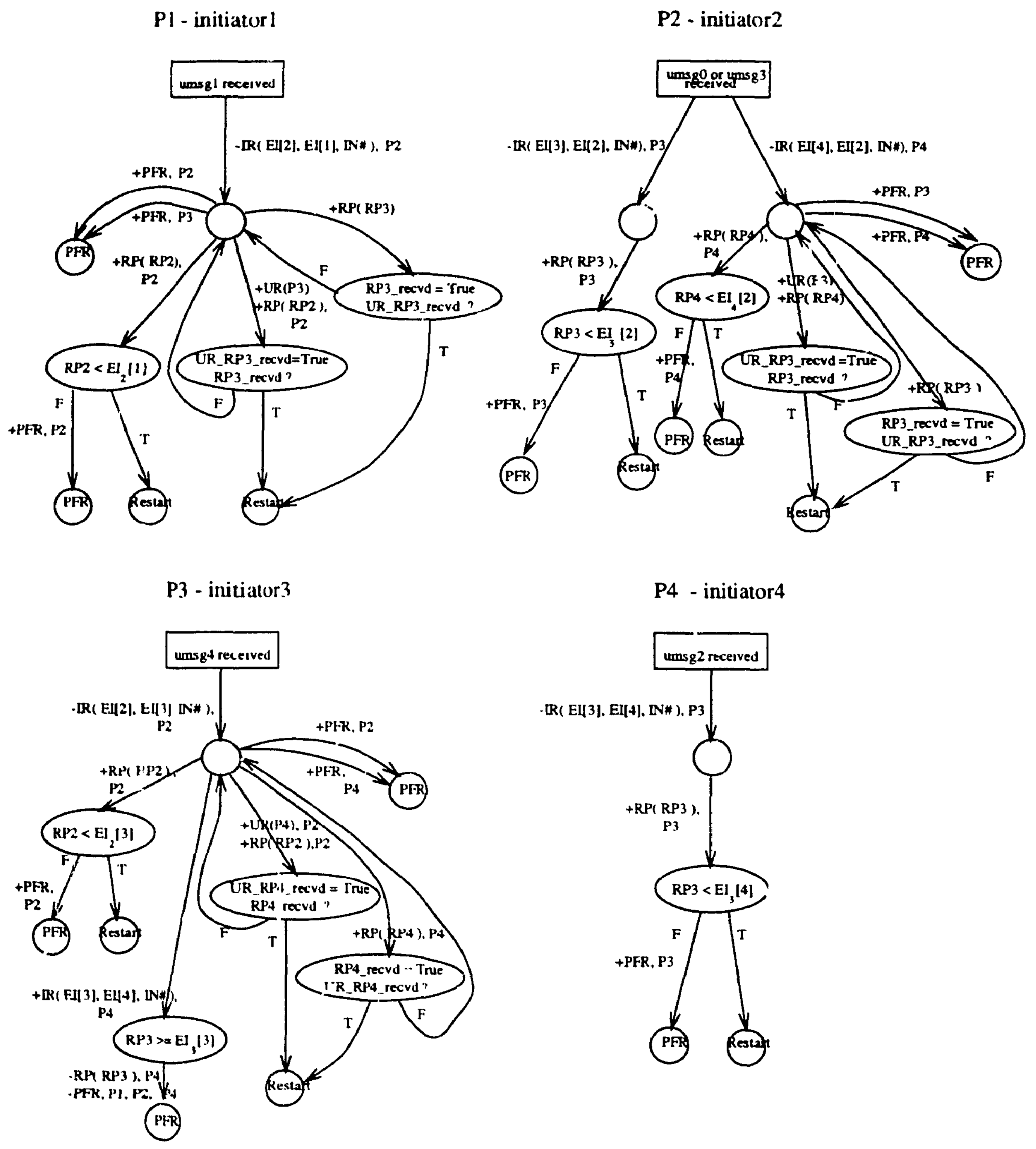
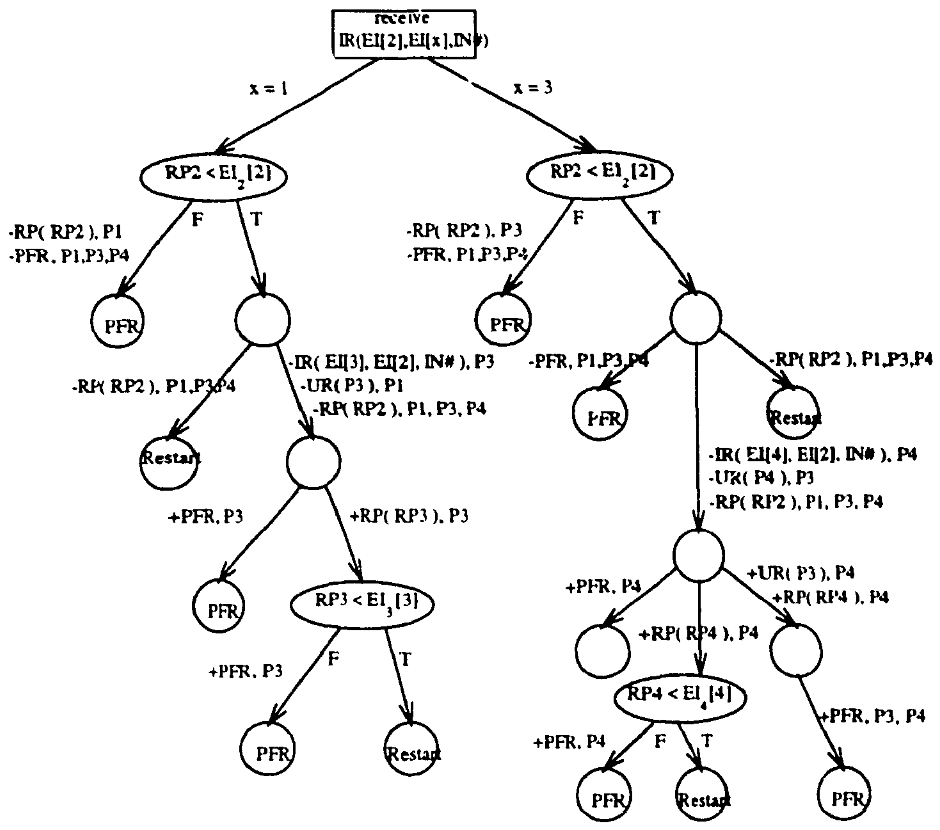
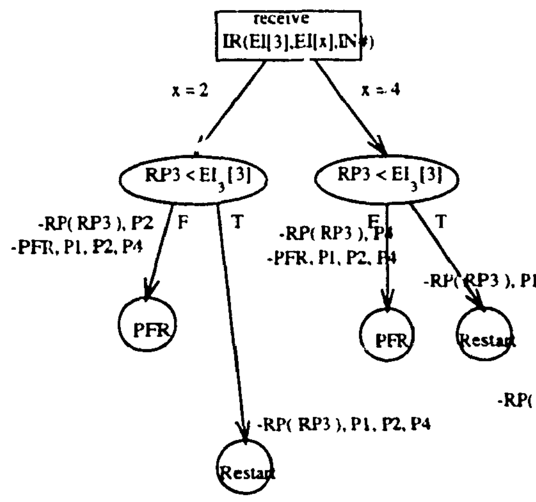


Figure 5.4a: Transition Diagrams for Initiator Processes

P2 - noninitiator2



P3 - noninitiator3



P4 - noninitiator4

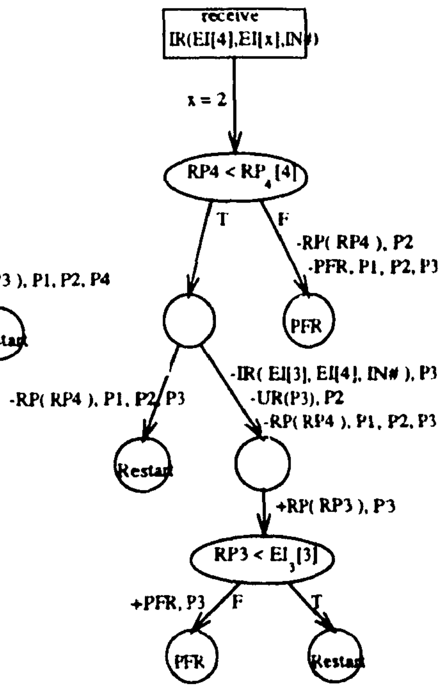


Figure 5.4b: Transition Diagrams for Non-Initiator Processes

RP_2 and if it is less than its current event index, it means that it has either received an earlier message *m*0 that was later detected as *umsg*0 during reexecution or it has wrongfully send message *m*1 as *umsg*1. In the later case, it just sends its *RP*(*RP*2) message to all other processes and restarts from its recovery point. In the former case, P_2 sends an *IR*(*EI*[3],*EI*[2],1) message to P_3 and an *UR*(*P*3) message to P_1 indicating that P_3 is in error state and process P_1 should receive a *RP*(*RP*3) message from P_3. P_2 also sends a *RP*(*RP*2) message to all other processes. It then waits for a *RP*(*RP*3) message or a *PFR* message from P_3. *RP*(*RP*3) message is received by all other processes if P_3 had indeed send *umsg*0 instead of *m*0 to P_2. If P_3 does not agree that it had send *umsg*0 and P_2 has wrongfully judged that, a *PFR* message is send indicating that the whole system has failed and should obtain a consistent state from its set of independent checkpoints.

Note that the flowcharts for the example of Figure 5.3 represent a substantial portion of the complete recovery procedure of Section 4.5.2.1 since each process in the system is not a sender or receiver of application messages to all other processes.

5.4.2. Validation of the Recovery Procedure

In this thesis Promela [Holz91] is the specification and modeling language, which is used to describe validation models that define the interactions of processes in a distributed system. The model is as simple as possible, yet sufficiently powerful to represent all types of coordination problems that can occur in distributed systems. The semantics of the language makes it possible to make a mapping from the flow chart language used in the transition diagrams of Section 5.4.1 to Promela programs straightforward.

The validation model is defined directly in terms of three specific types of objects: *processes*, message *channels* and state *variables*. All processes are global objects. Variables and channels represent data that can be either global or local to a process. In Promela, there is no difference between conditions and statements. Even isolated boolean conditions can be used as statements. The execution of a statement is conditional on its executability. All Promela statements are either executable or blocked, depending on the current values of variables or the contents of message channels. A process can wait for an event to happen by waiting for a statement to become executable.

A process has to be named, defined and instantiated to be executed. All types of processes that can be instantiated are defined in *proctype* declarations, which declare process behaviour but do not execute it. Initially, just one process is executed: a process of type *init*, which may start a number of other processes that will run concurrently with the *init* process. *Run* can be used in any process to spawn new processes, not just in the initial process. *Atomic* is used to indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It is used to reduce the complexity of a validation model.

Message channels are used to model the transfer the data from one process to another. They are declared as

```
chan p1_to_p2 = [5] of {byte, byte, byte, byte}
chan p2_to_p1 = [4] of {byte, byte, byte, byte};
chan p3_to_p1 = [4] of {byte, byte, byte, byte};
chan p2_to_p3 = [6] of {byte, byte, byte, byte};
chan p3_to_p2 = [6] of {byte, byte, byte, byte};
chan p2_to_p4 = [6] of {byte, byte, byte, byte};
chan p3_to_p4 = [3] of {byte, byte, byte, byte};
chan p4_to_p1 = [3] of {byte, byte, byte, byte};
```

```
    chan p4_to_p2 = [3] of {byte, byte, byte, byte};
    chan p4_to_p3 = [6] of {byte, byte, byte, byte};
```

that initializes, for example, channel *p1_to_p2* to store up to 5 messages, each consisting of four one-byte fields.

A message type definition of the form

```
mtype = {m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10,
    umsg, IR, PFR, UR, P1, P2, P3, P4, rm1, rm2, rm3, rm4};
```

makes the names of the constants, rather than the values, available to an implementation, which can improve error reporting.

Promela programs were written for the protocol example of Figure 5.3 corresponding to its transition diagrams of Figure 5.4. The Promela specification contains a process type *proctype* p1(), p2(), p3(), and p4() for each of the processes of the system, which are run atomically in the beginning of the execution of the program. These proctypes give the behaviour of the protocol indicating the messages received and messages sent over message channels that are defined globally. The message send is considered to be atomic with the correct message received to reduce the complexity of the validation model. As a first step, the application is run with no introduced errors, so that no error-recovery processes are activated. This verifies that the design of the application protocol is error-free. Next, the application is modified so that any sent messages can be replaced with a message of type *umsg*, and the application is re-run. If any of the process types receives a message of type *umsg*, it instantiates a copy of its initiator process type. This represents the fact that this process has detected failure, and has initiated recovery. Then, according to our recovery procedure, if the control message *IR* is received by any other process type it will initiate a copy of its non-initiator process type. This represents the fact that this process has not detected any error, but is participating in the recovery algorithm.

For example, if process type P1 receives an *umsg* from process type P2, it will start concurrently another process type *initiator1* which sends the *IR* message on its outgoing channel connecting P1 to P2. The process type P2 upon receiving the *IR* message, runs the process type *non_initiator2* and the rest of the recovery procedure is followed. There is no non_initiator1 process type since P1 is not a sender of an application message. Since Promela has the ability to model the manipulation of the contents of variables, the expressions such as $RP_i \geq EI_i(i)$ can be easily obtained and the corresponding path in the flow chart may be followed.

To validate a design, we need to be able to specify precisely what it means for a design to be correct. A design can be proven correct only with respect to specific correctness criteria. We are interested in finding the following standard criteria: the absence of deadlocks, logical incompleteness of the protocol specification (e.g., unspecified reception), improper terminations and unexecutable code segments.

The SPIN (Simple Promela INterpreter) automated protocol validation tool constructs, from the Promela model, a validator that can perform reachability analysis in three basic modes: random simulation, fully exhaustive state space search, or partial state space search. In particular, for our example, the controlled partial search technique named *supertrace* with the bit state space technique is used by the validators that are produced by SPIN. The validations using the supertrace mode can be performed in much smaller amounts of memory, and still retain excellent coverage of the state space.

Using SPIN in the supertrace mode it was found that there were no design errors with respect to deadlocks, unspecified receptions and improper terminations. The unused parts were eliminated from the specification, to include only the transitions given in the transition diagrams of Figure 5.4, to prevent extraneous detection of unreachable code from prematurely terminating the validation.

5.5. Summary

In this Chapter we have briefly discussed the various approaches followed to provide fault-tolerant protocols. The conventional design approach may require a central process to control the different sequences of state transitions to be followed for different error states. The semantic-based approach requires an expertise in the protocol to manually correct the errors against fault-tolerance and requires re-validation of the corrected protocol. Our approach to provide fault-tolerance to communication protocols is generic and does not require different state transitions for different error events or an expertise in the protocol. We specify the different sequences of state transitions that are based on our rollback recovery procedures of Chapter 4 using transition diagrams. These sequences are similar for any type of error events and their number depends on the location of the error in the whole system, whether the error resides directly in the sender process or indirectly in any other process related to the sender process. We have also proven the correctness of our sequences of state transitions by validating them using the modeling language Promela and automated protocol validation tool SPIN against the absence of any deadlocks, incompleteness, improper terminations and unspecified code segments.

CHAPTER 6

CONCLUSIONS

6.1. Conclusions

The goal of this research was to provide a unified approach to fault-tolerance in communication systems under a model of transient failures by formally incorporating them into the specification and design phases of the communication software development life cycle. Researchers have tended to address the issue of fault-tolerance in the area of distributed database and distributed computing by countering the effects of their individual causes. Not much work had been done in the past in the field of providing fault-tolerance to communication protocols, especially at the specification level. Since communication protocols include a large amount of abnormal processing triggered by transient failures, high reliability and performance in the presence of such events are required. To achieve this a generalized recovery procedure was developed whose states and transitions could be incorporated into the specification and design phases itself to provide fault-tolerance, to bring the system into a consistent state should a transient failure occur.

Our communication network is assumed to consist of a finite number of loosely coupled processors in a strongly connected fashion. These processors exchange messages over first-in-first-out communication channels that are assumed to be reliable and take a finite amount of delay to deliver the message. We also assume that each processor is equipped with a volatile storage and a stable storage to store necessary information required for recovery. The processes running on these processors are considered to be deterministic and non-fail-stop. The transient errors occurring in one process during the operation of the communication software may contaminate further checkpoints and local states of itself and other processes in the system. The operational errors are related to the environment and may change the global state of a system, but not its behaviour. It is assumed that these errors do not continue to occur. In the context of fault-tolerant protocols it is also assumed that the starting point in the protocol development life cycle is a complete specification and a design free of any syntactic and semantic errors.

In the literature we see conventional methods for design of real protocols where there is a specified central process to control the transmission and reception of messages for recovery from an error event, and different sequences of state transitions are followed for each state where the error event occurs. The real protocols often revert to an initial global state and all the state transitions from the initial global state are voided. The other approach we see is semantic-based where the specifications for protocols are flexibly designed first and then the errors against fault-tolerance are corrected manually. These design methods are application-specific and the modified protocol must be re-validated for syntactic and semantic correctness.

This thesis has proposed a unified method for design of fault-tolerant protocols for communication systems based on our checkpointing and rollback recovery procedures for protocols. The design methods described are application-independent. Since they require only the protocol structure and exchange of local information, they are regarded as unified design methods for any protocol and all failure modes. Any process where an error event occurs becomes an initiator process for recovery from the error event. The sequences of state transitions after each error event are automatically generated and are similar for all error events and for all non-initiator processes. This unified approach is independent of the inherent functions of any protocol. Any illegal sequence in the protocol can revert to an intermediate legal consistent state while retaining consistency in transmission and reception of messages.

Only the state transitions from an intermediate consistent state to the error event are voided. The provided recovery sequences of state transitions are validated to prove their correctness, using the specification and modeling language, Promela, and the Simple Promela INterpreter (SPIN), the automated protocol validation tool, to perform reachability analysis to detect any presence of deadlocks, logical incompleteness of the recovery procedure (e.g., unspecified reception), improper terminations and unexecutable code segments.

To obtain the required recovery procedures for protocols, several contributions were made towards our research. To start with, an algorithm is proposed to obtain the maximally reachable event index, MREI(i), which represents the event indices corresponding to sending/receiving messages that other processes must have reached before process P_i executes its event. This algorithm is then used by our other algorithmic procedures for checkpointing and rollback recovery in distributed systems based both on the pre-planned and the un-planned approaches. In the pre-planned approach, efficient checkpointing and rollback recovery procedures with their fault-tolerant versions are presented for non-fail-stop processes. These procedures meet most of the requirements for ideal checkpointing and recovery by obtaining the most recent and consistent possible system state for the process initiating the procedure, being non-intrusive, requiring a minimal number of processes to rollback, providing us with a maximum global consistent state after failures, being capable of recovering from any number of process failures in the system, and others. But a large stable storage is needed to record the local state along with other information after every transition by each process in the system. In the un-planned approach, efficient procedures for recovery are provided for non-fail-stop processes that do not require the application of checkpointing procedures, but uses contextual information exchanged during normal system progress and the independently saved checkpoint in stable storage and the saved message log in volatile storage. During recovery the processes need to replay only from their last correct independent checkpoint and not from their intial state. The recovery procedure meets all the requirements for an ideal recovery procedure stated in Section 2.3.1 by providing us with a maximum global consistent state; requiring only those processes to restart that are affected by error propagation and only those application messages to be retransmitted that are affected by error; and using a minimal number of recovery related messages. The procedure can be used to recover from any number of process failures in the system, including a total failure of all processes.

Our next contribution was to provide generalized recovery algorithms for fault-tolerance in communication protocols. The concepts are based on the algorithms provided for distributed systems, which are modified to take into account the effects of error propagation leading to some form of synchronization loss in the system. Error detection in the system is done by the reception of an erroneous message (unspecified message) by the receiver process that tries to find by re-execution from its known correct state if its own process state has corrupted or it is indeed a sender process that has failed by sending an unspecified message bringing the system to an inconsistent state. Our procedure also considers the case when a receiver process is not able to detect the error since its own process state was corrupted and it considered the erroneous message received as correct, but later in the protocol execution some other process is able to detect system failure. Our procedure allows concurrent processes detecting errors and therefore concurrent initiations of the recovery procedure for non-fail-stop processes. The stable storage requirement is the minimum for saving independent checkpoints, and volatile storage is used for logging messages received to reduce the time required to access them during re-execution. Depending on the error event, some of the processes may not be required to roll back, meaning that our procedure forces only the necessary and minimal number of processes to roll back. A message will be retransmitted only if it was corrupted keeping the number of message retransmissions to a minimum.

Our algorithms have been compared with other algorithms provided for recovery in the literature for protocols, and it is found that they are better than them in all the features of an ideal recovery procedure.

6.2. Recommendations for Future Work

In our future research, we would like to consider the other causes that lead to the loss of coordination within processes, which were not considered in our research. These causes include inconsistent initialization, where different processes are initialized to local states that are inconsistent with one another, and transmission errors, where messages may be lost, corrupted or reordered during transmission.

We would also like to study recovery in non-deterministic systems. Since we have messages logged on storage, and these messages are re-executed from the last independent checkpoint free from errors, we assume the processes to be deterministic from the point of re-execution until the state before the error state.

It would also be a good idea to make the recovery procedures for the un-planned approach to be fault-tolerant. We would also like to move our research further into the area of responsive communication protocols, to consider the real time features of our fault-tolerant protocols. We would then like to consider some standard protocols to check their responsiveness.

LIST OF REFERENCES

[BBG83] A. Borg, J. Baumbach and S. Glazer, *"A Message System Supporting Fault Tolerance,"* Proc. of the 9th ACM Symp. on Operating Systems Principles, Oct. 1983, pp 90-99.

[BCS84] D. Briatico, A. Ciuffoletti and L. Simoncini, *"A Distributed Domino-effect free Recovery Algorithm,"* Proc. of the Symposium on Reliability in Distributed Software and Database Systems, Oct. 1984, pp 207-215.

[Boch78] G.v. Bochmann, *"Finite State Description of Communication Protocols,"* Computer Networks, Vol.2, No.4/5, Sept-Oct 1978, pp 361-372.

[ChKr88] M.J. Chung and M.S. Krishnamoorthy, *"Algorithms of Placing Recovery Points,"* Information Processing Letters, Vol.28, 1988, pp 177-181.

[ChLa85] K. Chandy and L. Lamport, *"Distributed Snapshots: Determining Global States of Distributed Systems,"* ACM Trans. on Computer Systems, Vol.3, No.1, Feb 85, pp 63-75.

[Dijk74] E.W. Dijkstra, *"Self-Stabilizing systems in spite of Distributed Control,"* Communications of the ACM, Vol.17, No.11, Nov. 1974, pp 643-644.

[ElZw92] E.N. Elnozahy and W. Zwaenepoel, *"Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit,"* IEEE Trans. on Computers, Vol.41, No.5, May 1992, pp 526-531.

[FoZw90] J. Fowler and W. Zwaenepoel, *"Causal Distributed Breakpoints,"* Proc. of the 10th International Conf. on Distributed Computing Systems, June 1990, pp 134-141.

[FrTa89] T.M. Frazier and Y. Tamir, *"Application Transparent Error Recovery Technique for Multicomputers,"* Proc. of the Fourth Conference on Hypercubes, Concurrent Computers & Applications, Vol.1, Mar 89, pp 103-108.

[Gamb92] D. Gambhir, *Local Directed Graphs*, Ph.D. Thesis, Polytechnic Univ., Brooklyn, N.Y., 1992.

[GaFr91] D. Gambhir and I.T. Frisch, *"Automated Communication Network Software Fault Isolation,"* Proc. of the IEEE Intl. Conf. on Systems, Man and Cybernetics, Oct 13-16, Vol.2, 1991, pp 719-724.

[GoMu91] M.G. Gouda and N.J. Multari, *"Stabilizing Communication Protocols"* IEEE Trans. on Computers, Vol.40, No.4, Apr 1991, pp 448-458.

[Holz91] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991

[IsMo89] S. Israel and D. Morris, *"A Non-intrusive Checkpointing Protocol,"* Proc. of the IEEE Intl. Phoenix Conference on Computers & Communications, 1989, pp 413-421.

[JoZw90] D.B. Johnson and W. Zwaenepoel, *"Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing,"* Journal of Algorithms, Vol.11, No.3, Sep 1990, pp 462-491.

[Kak91] Y. Kakuda, *A Recovery Sequence Generation System for Design of Recoverable Protocols,"* IEICE Trans. on Information & Systems, Vol.E74, No.6, June 1991, pp 1715-1727.

[KaKi91] Y. Kakuda and T. Kikuno, *"Verification of Responsiveness for Communica-*

tion Protocols," IEICE Japan, Tech. Group Paper, CPSY 91-58 (FTS 91-57), Dec 1991.

[KaKi92] Y. Kakuda and T. Kikuno, *"Issues in Responsive Protocols Design,"* Proc. of the Second International Workshop on Responsive Computer Systems, Saitama, Japan, Oct 1-2, 1992, pp 8-12.

[KKMS91] Y. Kakuda, T. Kikuno, M. Malek and H. Saito, *"Efficient Checkpointing for Protocol Recovery in Communication Systems,"* Proc. of the 1991 Intl. Symp. on Communication (ISCOM '91), Tainan, Taiwan, Dec 1991, pp 704-707.

[KKMS92] Y. Kakuda, T. Kikuno, M. Malek and H. Saito, *"A Unified Approach to Design of Responsive Protocols,"* Proc. of the 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, Amherst, Massachusetts, July 1992, pp 8-15.

[KoTo87] R. Koo and S. Toueg, *"Checkpointing and Rollback-Recovery for Distributed Systems,"* Trans. on Software Engg., Vol.SE-13, No.1, Jan 1987, pp 23-31.

[Lam78] L. Lamport, *"Time, Clocks and the Ordering of Events in a Distributed System,"* Communications of the ACM, 21 (7), 1978, pp 558-565.

[LeAn90] P.A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, Springer Verlag, 1990.

[LeBh88] P. Leu and B. Bhargava, *"Concurrent Robust Checkpointing and Recovery in Distributed Systems,"* Proc. of the Intl. Conference on Data Engineering, Feb. 88, pp 154-163.

[Mal90] M. Malek, *"Responsive Systems: A Challenge for the Nineties,"* Proc. EUROMICRO'90, 16th Symposium on Microprocessing and Microprogramming, Keynote Address, Amsterdam, Microprocessing and Microprogramming, 30, Aug 1990, pp 9-16.

[Mili90] A. Mili, *An Introduction to Program Fault Tolerance*, Prentice Hall, 1990.

[Mult89] N.J. Multari, *Towards a Theory for Self-Stabilizing Protocols*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Texas at Austin, 1989.

[PoPr83] M.L. Powell and D.L. Presotto, *Publishing: A reliable Broadcast Communication Mechanism,"* Proc. of the 9th ACM Symp. on Operating Systems Principles, Oct. 1983, pp 100-109.

[PrSa91] Probert, P. and Saleh, K. *"Synthesis of Communication Protocols: Survey and Assessment,"* IEEE Trans. on Computers, Vol.40, No.4, Apr 1991, pp 468-476.

[Raynal88] M. Raynal, *Networks and Distributed Computation*, The MIT Press, 1988.

[RaSi96] Michel Raynal and Mukesh Singhal, *"Logical Time: Capturing Causality in Distributed Systems,"* Computers, Feb 1996, pp 49-56.

[RLT78] B. Randell, P.A. Lee and P.C. Treleaven, *"Reliability Issues in Computing System Design,"* ACM Computing Surveys, Vol.10, No.2, Oct 1978, pp 123-165.

[SAAA94] Kassem Saleh, Imtiaz Ahmed, Khaled Al-Saqabi and Anjali Agarwal, *"An Efficient Recovery Procedure for Fault-Tolerance in Distributed Systems,"* Journal of Systems and Software, 1994, pp 39-50.

[SAAA95] K. Saleh, I. Ahmed, K. Al-Saqabi, and A. Agarwal, *"A Recovery Approach to the Design of Stabilizing Communication Protocols,"* Journal of Computer Communications, Vol. 18, No. 4, April 1995, pp 276-287.

[SaAg93a] K. Saleh and A. Agarwal, *"Efficient and Fault-Tolerant Checkpointing Procedures for Distributed Systems,"* Proc. of the IEEE Intl. Phoenix Conference on Computers & Communication (IPCCC-93), Mar. 1993, pp 161-167.

[SaAg93b] K. Saleh and A. Agarwal, *"Efficient and Fault-Tolerant Checkpointing and Recovery Procedures for Distributed Systems,"* Networking and Distributed Computing Journal, Vol.3, No.2, 1993, pp 169-185.

[Sarr93] A.J. Sarraf, *Checkpointing and Rollback Recovery in a non-FIFO Multichannel Distributed Environment,* Master's Thesis, Dept. of Computer Science, Concordia University, June 1993.

[Schn93] M. Schneider, *"Self-Stabilization,"* ACM Computing Surveys, Vol.25, No.1, Mar 1993, pp 45-67.

[SiWe89] A.P. Sistla and J.L. Welch, *"Efficient Distributed Recovery Using Message Logging,"* Proc. 8th annual ACM Symp. Principles of Distributed Computing, Aug 1989, pp 223-238.

[StYe85] R.E. Strom and S.A. Yemini, *"Optimistic Recovery in Distributed Systems,"* ACM Trans. Comput. Syst., Vol.3, No.3, Aug 1985, pp 204-226.

[SUA94] K. Saleh, H. Ural and A. Agarwal, *"A Modified Distributed Snapshots Algorithm for Stabilizing Protocols"* to appear in the Journal of Computer Communications

[TKT89] Z. Tong, R.Y. Kain and W.T. Tsai, *"A Lower Overhead checkpointing and Rollback Recovery Scheme for Distributed Systems,"* Proc. of the 8th Symposium on Reliable Distributed Systems, Oct. 89, pp 12-20.

[Venk88] K. Venkatesh, *Global States of Distributed System: Classification and Applications*, Ph.D. Thesis, Dept. of Computer Science, Concordia University, Montreal, Jan 1988.

[VRL87] K. Venkatesh, T. Radhakrishnan and H.F. Li, *"Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery,"* Information Processing Letters, Vol.25, No.5, July 1987, pp 295-303.