View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ethesis@nitr

ENHANCED INDEX BASED CHECKPOINTING ALGORITHM FOR DISTRIBUTED SYSTEMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

In

Computer Science and Engineering

By

E. SRIKANTH

Department of Computer Science & Engineering
National Institute of Technology
Rourkela

2007

ENHANCED INDEX BASED CHECKPOINTING ALGORITHM FOR DISTRIBUTED SYSTEMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
In
Computer Science and Engineering

By

E. SRIKANTH

Under the Guidance of

Dr D. P. MOHAPATRA

Department of Computer Science & Engineering
National Institute of Technology
Rourkela

2007

National Institute of Technology
Rourkela

CERTIFICATE

This is to certify that the thesis entitled, **An Enhanced Index-Based Checkpointing Algorithm for Distributed Systems** submitted by Sri **E. Srikanth** in partial fulfillment of the requirements for the award of Master of Technology Degree in Computer Science and Engineering at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date:

Dr. D. P. Mohapatra
Assistant Professor
Dept. of Computer Science and Engg.
National Institute of Technology
Rourkela – 769008

ACKNOWLEDGEMENT

I would like to thank my thesis supervisor, **Dr. D. P. Mohapatra**, for introducing me to Distributed Systems. This thesis work was enabled and sustained by his vision and ideas. His scholarly guidance and invaluable suggestions motivated me to complete my thesis work successfully.

I would take this opportunity to express my gratitude to Professor Dr. S. K. Jena, Head of the Department of Computer Science & Engg for his valuable suggestions and cooperation at various stages.

I am also thankful to all the faculty members, Department of Computer Science and Engg. who gave all possible help to bring my thesis work to the present shape.

I would like to take the chance to express my appreciation to my parents. Their continuous love and support gave me strength for pursuing my dream.

I would also like to thank my friends who have been a source of encouragement and inspiration throughout the duration of this thesis.

Finally, I thank one and all who helped me directly and indirectly in the completion of this work.

(E. Srikanth)

CONTENTS

1 **Introduction** **1**

1.1 Motivation........ 4

1.2 Thesis Objective........ 5

1.3 Thesis Organization........ 5

2 **Overview Of Checkpointing** **6**

2.1 Aspects of Checkpointing........ 6

2.1.1 Frequency of Checkpointing........ 6

2.1.2 Contents of a Checkpoint........ 7

2.1.3 Methods of Checkpointing........ 7

2.2 Overhead of Checkpointing........ 7

2.3 Consistent System State........ 8

2.4 Issues in Checkpointing........ 9

2.4.1 Orphan Messages and Domino Effect........ 9

2.4.2 Lost Messages........ 10

2.4.3 Livelocks........ 11

2.5 Types of Checkpointing Protocols........ 12

2.5.1 Asynchronous Checkpointing Protocols........ 12

2.5.2 Synchronous Checkpointing Protocols........ 13

2.5.3 Quasi-Synchronous Checkpointing Protocols........ 16

2.5.4 Log based Recovery Protocols........ 18

3 **Index Based Checkpointing Protocols** **22**

3.1 Model of the Distributed Computation........ 22

3.2 BCS........ 24

3.3 Lazy-BCS-Aftersend........ 25

3.4 BQF........ 26

3.4.1 Equivalence between Checkpoints........ 27

3.4.2 Tracking the Equivalence Relation On-The-Fly........ 28

3.4.3 The BQF Algorithm........ 30

4 Enhanced Index Based Checkpointing Protocol **33**

4.1 Simulation Environment.. 34

4.2 SPIN... 35

4.2.1 The structure.. 37

4.3 The Algorithm.. 38

4.4 Drawbacks of BQF.. 40

4.5 Experimental Results.. 40

5 Conclusion and Future Work **43**

References **44**

ABSTRACT

Rollback-recovery in distributed systems is important for fault-tolerant computing. Without fault tolerance mechanisms, an application running on a system has to be restarted from scratch if a fault happens in the middle of its execution, resulting in loss of useful computation. To provide efficient rollback-recovery for fault-tolerance in distributed systems, it is significant to reduce the number of checkpoints under the existence of consistent global checkpoints in index-based distributed checkpointing algorithms. Because of the dependencies among the processes states that induced by inter-process communication in distributed systems, asynchronous checkpointing may suffer from the domino effect. Therefore, a consistent global checkpoint should always be ensured to restrict the rollback distance.

The quasi-synchronous checkpointing protocols achieve synchronization in a loose fashion. Index-based checkpointing algorithm is a kind of typical quasi-synchronous checkpointing mechanism. The algorithm proposed in this thesis follows a new strategy to update the checkpoint interval dynamically as opposed to the static interval used by the existing algorithms explained in the previous chapter. Whenever a process takes a forced checkpoint due to the reception of a message with sequence number higher than the sequence number of the process, the checkpoint interval is either reset or the next basic checkpoint is skipped depending on when the massage has been received.

The simulation is built on SPIN, a tool to trace logical design errors and check the logical consistency of protocols and algorithms in distributed systems. Simulation results show that the proposed scheme can reduce the number of induced forced-checkpoints per message 27-32% on an average as compared to the traditional strategies.

LIST OF FIGURES

2.1 An example of Consistent and Inconsistent State... 9
2.2 Effects of Orphan Messages and Domino Effect... 10
2.3 Lost Messages... 11
2.4 State before Livelock.. 11
2.5 Livelock Situation... 11
2.6 Domino Effect.. 12
2.7 *Z*-path determination... 16
2.8 Orphan process.. 20

3.1 BCS
a. P_i does not increase its index.. 24
b. P_i must increment its index... 24
3.2 Examples of pairs of equivalent checkpoints... 27
3.3 Upon the receipt of m`, P_1 detects $C_{1, sn, 0} \overset{L_{sn}}{\equiv}$ next $(C_{1, sn, 0})$............................ 28

4.1 The structure of SPIN simulation and verification.. 38
4.2 None faster situation.. 41
4.3 One faster situation.. 42

Chapter 1

INTRODUCTION

Motivation
Thesis Objective
Thesis Organization

1. INTRODUCTION

A *distributed system* is composed of a set of machines which do not share a global clock, the machines communicate with each other by exchanging messages over a communication network. Each machine in the distributed system has its own memory and runs its own operating system. The machines in a distributed system offer their resources for collective computation. The resources owned and controlled by a machine are said to be *local* to it, while the resources owned and controlled by other computers and those that can only be accessed through the network are said to be *remote*. These resources can be of various types such as computation nodes, storage devices etc. A large number of applications have been developed to harness the power of distributed systems.

Typically a distributed system has the following characteristics:

- *Multiple nodes* – A distributed system is composed of multiple independent nodes belonging to different computers, not merely multiple processors on the same computer.
- *Heterogeneity* – The nodes in a distributed system may consist of machines having different architectures and possibly running different types of operating systems.
- *Message passing* – Processes on the different resource nodes may communicate using diverse networking protocols over different networking technologies. Therefore, the characteristics of the underlying communication links can be different. The nodes in most distributed systems are reachable from one another.
- *Concurrency* – Each of the nodes in a distributed system provides independent functionality, and operates concurrently with other nodes
- *Decentralized control* – No single computer is necessarily responsible for configuration, management, or policy control for the whole distributed system. However, some functionality may reside in a central node or a set of nodes by necessity.
- *Openness* – Many distributed systems are *open,* i.e., an unbounded number of nodes or components can be added or changed even while the system is running.

The main objective of a distributed system is to achieve high throughput for distributed applications through concurrent computation and to increase accessibility to resources not commonly available to a single machine.

Advantages of distributed systems are:

- *Higher price/performance ratio* – By interconnecting powerful workstations with high speed communication network we achieve higher performance at lower cost.
- *Resource sharing* – A node in a distributed system can access both software and hardware resources of another node remotely over the communication network.
- *Improved availability* – A distributed computing system provides improved reliability and availability because a few components of the system can fail without affecting the availability of the rest of the system.
- *Improved reliability* – By replicating data and services the distributed systems can be made fault tolerant

Distributed systems have been used for a wide variety applications ranging from scientific simulations collaborative engineering, supercomputer enabled scientific instruments, applications in Geographical Information Systems (GIS) like weather prediction, railway or airline reservation systems etc.

Distributed systems are composed of multiple computing resources connected by communication links. Since failure of nodes and links are assumed to be independent, larger the system, higher is its probability of failure. Therefore, in a distributed system, failures are relatively common events. Distributed systems should remain at least partially available and functional even if some of their nodes or communication links fail or misbehave. Without fault tolerance mechanisms, the system and the applications running on it need to be restarted every time a failure occurs. Many of the distributed applications mentioned above are long running, taking hours or even days in some cases to complete. If a fault occurs in the middle of a long running application, long hours of useful computation will be lost. Thus an application may take a long time to complete in the presence of such failures. Fault tolerance techniques can allow applications to run to completion in the presence of faults with minimal disruption. Since such techniques do not need an application to restart when a fault occurs, they also save system resources and improve system throughput.

Fault tolerance in distributed systems is usually achieved by some form of redundancy. Two forms of redundancy are normally used in distributed systems to provide tolerance to node failures.

- *Spatial redundancy* involves processes replicated on several computing nodes. Examples of such redundancy are TMR (Triple Modular Redundancy), voter based NMR etc. This type of redundancy is important for mission critical or hard real-time applications where failure is very critical and in-time recovery must be guaranteed. Spatial redundancy requires extra hardware and therefore is costly, but the reaction time to a fault is very fast.
- *Temporal redundancy* also called backward error recovery, is to re-execute a job, or part of a whole job, when a fault happens. This kind of redundancy is particularly suitable for non-critical applications and implementation of such a scheme does not demand extra hardware. Checkpoint and recovery protocols are popularly used for providing fault tolerance through temporal redundancy in message passing distributed systems.

Fault tolerance through checkpoint and recovery technique includes taking checkpoint of an application process periodically, and logging the checkpoint in a stable storage which is immune to failures. Checkpoint of an application process is the information about the state of the process and can be used to restart it from a state corresponding to the checkpoint. On the other hand, rollback recovery for an application is defined as the procedure for restarting the application process from a checkpoint stored in stable storage. For an application involving a single process, checkpoint and recovery is simple. In the event of a node failure, the application process is restarted from its latest checkpoint, i.e., rolled back to its last checkpoint .This technique saves re-computation till the latest checkpoint. However, several issues with checkpoint and recovery technique arise in a distributed system which runs applications having multiple concurrent processes, communicating with each other via messages.

1.1 MOTIVATION

Checkpoint/restart provides two main functions for an operating system. First, checkpoint/restart is a mechanism for fault tolerance. Applications may be checkpointed periodically (or based on notifications from fault monitors). Once the application state has been committed to stable storage, the application may be restarted and reconfigured to work around the fault. Second, checkpoint/restart may be used as a mechanism for preemption. This form of preemption is useful in environments where virtual memory is scarce. In such

environments the operating system is unable to allocate sufficient memory to hold all runnable processes, so checkpoint/restart is used to save the application state to the file system.

Checkpoint and restart techniques allow programs to make progress in the face of recurring system downtime. The program state is periodically recorded on stable storage during execution. In the event of a system failure, the computation may be restarted and continued from the `frozen' state on disk. Several factors motivate the need for a general-purpose checkpoint and restart mechanism for (massively) parallel computing systems. Parallel computers have traditionally been employed to satisfy the needs of science and engineering applications at the edges of computational feasibility. Such applications typically have long running times, ranging from several hours to days at a time, and thus require the systems they run on to provide long periods of continuous failure-free time. In addition, these applications tie up the systems resources preventing other less demanding applications from running, thereby decreasing system throughput and increasing average turnaround time. The increase in the number of system components causes the mean time between failures of the system to decrease. Parallel programs must also successfully contend with this increase in the number of points of failure. There are exacting demands on the capability of parallel computing systems to execute long running, compute intensive tasks to completion.

Checkpoint and restart techniques endow parallel programs with a higher degree of fault tolerance than otherwise possible thus enabling complete runs despite non-continuous failure-free time. These facilities provide a number of other benefits in the context of parallel computing systems. Large compute intensive tasks may be checkpointed allowing other programs to run, thereby increasing the average turnaround time of the system. Programs may execute on different machines during a single run, and also `shrink' and `expand' being checkpointed after running for a while on a system with a large number of processors, then continuing on smaller, more easily available systems, and vice versa

1.2 THESIS OBJECTIVE

To provide efficient rollback-recovery for fault-tolerance in distributed systems, it is significant to reduce the number of checkpoints under the existence of consistent global checkpoints in index-based distributed checkpointing algorithms.

The main objective of our research work is to develop an efficient checkpointing algorithm that would minimize the number of checkpoints under the existence of consistent global checkpoints.

1.3 THESIS ORGANIZATION

This thesis report consists of 5 chapters. Chapters 2 gives an overview of checkpointing, where the aspects of checkpointing, overhead incurred in taking checkpoints and different types of checkpointing algorithms have been discussed briefly. In Chapter 3 the existing index based checkpointing algorithms, BCS, Lazy-BCS-Aftersend, and BQF have been discussed. In Chapter 4 the proposed algorithm, Enhanced index based checkpointing algorithm has been discussed. It also points out the drawbacks of the existing techniques and how they are overcome in this proposed algorithm. Conclusion and future work are discussed in Chapter 5.

Chapter 2

OVERVIEW OF CHECKPOINTING

Aspects of Checkpointing

Overhead of Checkpointing

Consistent System State

Issues in Checkpointing

Different Types of Checkpointing Protocols

2. Overview Of Checkpointing Protocols

Checkpoint is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. Checkpointing is the process of saving the status information. A checkpoint of a process is the information about the state of a process at some instant of time. Fault tolerance through checkpoint and recovery techniques includes taking check- point of an application process periodically, and logging the checkpoint in a stable storage which is immune to failures. Checkpoint of an application process is the information about the state of the process and can be used to restart it from a state corresponding to the checkpoint. On the other hand, roll back recovery for an application is defined as the procedure for restarting the application process from a checkpoint stored in stable storage.

A checkpoint can be saved on either stable storage or the volatile storage of another process, depending on the failure scenarios to be tolerated. For long-running scientific applications, checkpointing and rollback-recovery can be used to minimize the total execution times in the presence of failures.

For an application involving a single process checkpoint and recovery is simple. In the event of a node failure, the application process is restarted from latest checkpoint, i.e. rolled back its latest checkpoint. However, several issues with checkpointing and recovery arise in distributed system which runs applications having multiple concurrent processes, communicating with each other via messages.

2.1 Aspects of Checkpointing

Some of the aspects to be considered with checkpointing are frequency of checkpointing, contents of a checkpoint and methods of checkpointing [17].

2.1.1 Frequency of Checkpointing

A checkpointing algorithm executes in parallel with the underlying computation. Therefore, the overheads introduced due to checkpointing should be minimized. Checkpointing should enable a user to recover quickly and not lose substantial computation in case of an error, which necessitates frequent checkpointing and consequently significant

overhead. The number of check-points initiated should be such that the cost of information loss due to failure is small and the overhead due to checkpointing is not significant. These depend on the failure probability and the importance of the computation. For example, in a transaction processing system where every transaction is important and information loss is not permitted, a checkpoint may be taken after every transaction, increasing the checkpointing overhead significantly.

2.1.2 Contents of a Checkpoint

The state of a process has to be saved in stable storage so that the process can be restarted in case of an error. The state context includes code, data and stack segments along with the environment and the register contents. Environment has the information about the various files currently in use and the file pointers. In case of message-passing systems, environment variables include those messages which are sent and not yet received.

2.1.3 Methods of checkpointing

The methodology used for checkpointing depends on the architecture of the system. Methods used in multiprocessor systems should incorporate explicit coordination unlike uniprocessor systems. In a message-passing system, the messages should be monitored and if necessary saved as part of the global context. The reason is that the messages introduce dependencies among the processors. On the other hand, a shared memory system communicates through shared variables which introduce dependency among the nodes and thus, at the time of checkpointing, the memory has to be in a consistent state to obtain a set of concurrent checkpoints.

2.2 OVERHEAD OF CHECKPOINTING

During a failure-free run, every global checkpoint incurs coordination overhead and context saving overhead in a Distributed system. We define them as follows.

- **Coordination Overhead**

  In a distributed system, coordination among processes is needed to obtain a consistent global state. Special messages and piggy-backed information with regular messages are used to obtain coordination among processes. Coordination overhead is due to these special messages and piggy-backed information.

- **Context Saving Overhead**

  The time taken to save the global context of a computation is defined as the context-saving overhead. Overhead for checkpoint storage in stable storage contributes a major part of the overhead of checkpoint and recovery protocols. This overhead is proportional to the size of the context.

2.3 CONSISTENT SYSTEM STATE

In distributed system, a computation node can take checkpoints of its local processes only, and such checkpoints are called *local checkpoints* [8]. A *global checkpoint* of a distributed system is defined as set of local checkpoints, one from each of its processes in the system. After recovery the system must be in a *consistent state*. A global state of a message-passing system is a collection of the individual states of all participating processes and of the states of the communication channels. Intuitively, a consistent global state is one that may occur during a failure-free, correct execution of a distributed computation. More precisely, a consistent system state is one in which if a process state reflects a message receipt, then the state of the corresponding sender reflects sending that message.

For example, Fig 2.1 shows two examples of global states, a consistent state in Fig 2.1(a), and an inconsistent state in Fig 2.1(b). Note that the consistent state in Fig 2.1(a) shows message m_1 to have been sent but not yet received. This state is consistent, because it represents a situation in which the message has left the sender and is still traveling across the network. On the other hand, the state in Fig 2.1(b) is inconsistent because process P_2 is shown to have received m_2 but the state of process P_1 does not reflect sending it. Such a state is impossible in any failure-free, correct computation.

Inconsistent states occur because of failures. For example, the situation shown in part (b) of Figure 2.1 may occur if process P_1 fails after sending message m_2 to P_2 and then restarts at the state shown in the figure. A fundamental goal of any rollback-recovery protocol is to bring the system into a consistent state when inconsistencies occur because of a failure. The reconstructed consistent state is not necessarily one that has occurred before the failure. It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free, correct execution.

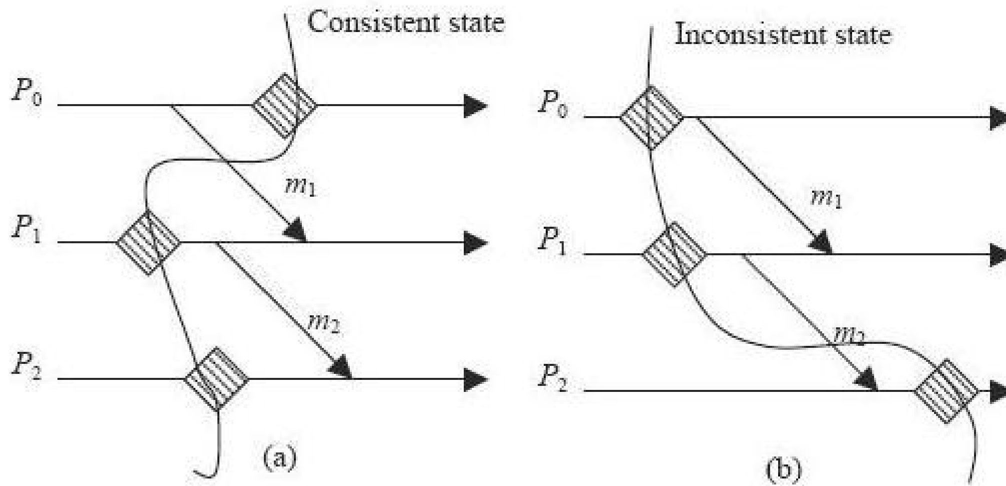


Fig 2.1: An example of a consistent and inconsistent state.

2.4 ISSUES IN CHECKPOINTING

In concurrent systems, several processes cooperate by exchanging information to accomplish task. The information exchanges through the message passing. In such system, if one of the cooperating processes fails and resumes execution from a recovery point, then the effects it has caused at other processes due to the information it has exchanged with them after establishing the recovery point will have to be undone. To undo the effects caused by a failed process at an active process, the active process must also rollback to an earlier state. Rolling back processes in concurrent system is more difficult than in the case of a single process. The following discussion illustrates how the rolling back of processes can cause further problems [27].

2.4.1 Orphan Messages and Domino Effect

Consider the three processes X, Y and Z are running concurrently in Fig 2.2. The parallel lines are showing the executions of the processes. These processes are communicated through exchange of messages. Each symbol **'['** marks a recovery point to which process can be rolled back in the event of a failure.

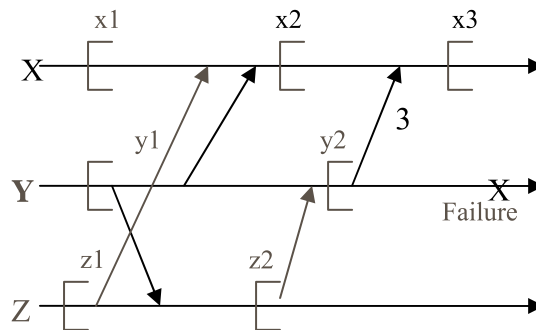


Fig 2.2: Effects of Orphan Messages and Domino Effect

If the process X is to be rolled back, it can be rolled back to the recovery point x_3 without affecting any other process. Suppose that Y fails after sending message *m* and is rolled back to y_2. In this case, the receipt of *m* is recorded in x_3, but the sending of *m* is not recorded in y_2. Now we have a situation where X has received message *m* from Y, but Y has no record of sending it, which corresponds to an inconsistent state. Under such circumstances, *m* is referred to as an *orphan* message and process X must also rollback. X must roll back because Y interacted with X after establishing its recovery point y_2. When Y is rolled back to y_2, the event that is responsible for the interaction is undone. Therefore, all the effects at X caused by the interaction must also be undone. This can be achieved by rolling back X to recovery point x_2. Likewise, it can be seen that, if Z is rolled back, all three processes must rollback to their very first recovery points, namely, x_1, y_1, z_1. This effect, where rolling back one process causes one or more processes to rollback, is known as the *domino effect* and orphan messages are the cause.

2.4.2 Lost Messages

Suppose that checkpoints x_1 and y_1 in Fig 2.3 are chosen as the recovery points for processes X and Y, respectively. In this case, the event that sent message *m* is recorded in x_1, while the event of its receipt at Y is not recorded in y_1. If Y fails after receiving message *m* the system is restored to state x_1, y_1, in which message *m* is *lost* as process X is past the point where it sends message *m*. This condition can also arise if *m* is lost in the communication channel and processes X and Y are in state x_1 and y_1, respectively.

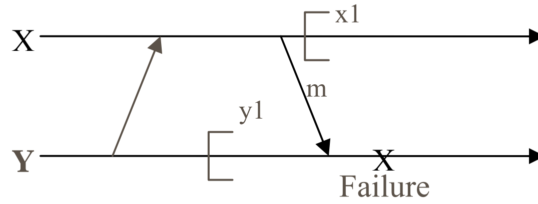


Fig 2.3: Lost Messages

2.4.3 Livelocks

In rollback recovery, livelock is a situation in which a single failure can cause an infinite number of rollbacks, preventing the system from making process. A livelock situation in a distributed system is shown in Fig 2.4. Fig 2.4 illustrates the activity of two processes X and Y until the failure of Y. Process Y fails before receiving message n_1, sent by X. When Y rollbacks to y_1, there is no record of sending message m_1, hence X must rollback to x_1.When process Y recovers, it sends out m_2 and receives n_1 (Fig 2.5). Process X, after resuming from x_1, sends n_2 and receives m_2. However, because X rolled back, there is no record of sending n_1 and hence Y has to rollback for the second time. This forces X to rollback too, as it has received m_2, and there is no record of sending m_2 at Y. This situation can repeat indefinitely, preventing the system from making any progress.

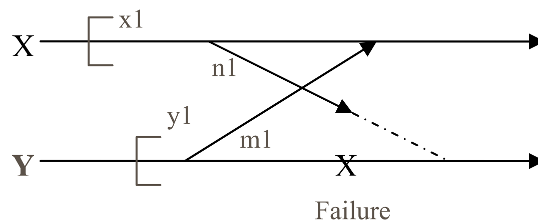


Fig 2.4: State before Livelock

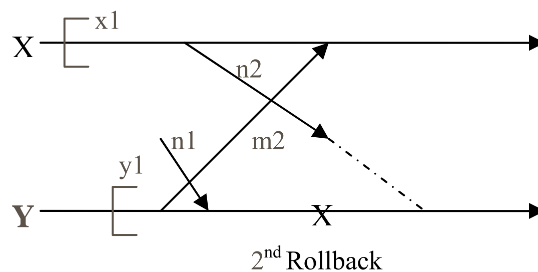


Fig 2.5: Livelock Situation

2.5 Types of Checkpointing Protocols

Several checkpointing and recovery protocols are available in the literature [11]. These protocols are classified into following categories. (i) Asynchronous checkpointing protocols (ii) Synchronous checkpointing protocols (iii) Quasi-Synchronous checkpointing protocols (iv) Log-based recovery protocols

2.5.1 Asynchronous Checkpointing Protocols

The protocols in this class allow taking local checkpoints independent of other processes in the distributed system. Such protocols are also referred as uncoordinated or asynchronous checkpointing protocols. The fault-free runtime overhead is least for these kinds of protocols because no coordination is needed between the processes to take checkpoints. During recovery, processes coordinate among themselves to determine a consistent global state. Therefore, recovery overhead is high. Due to bad placement of checkpoints over the communication pattern, the recovery protocol may require several rounds of coordination and rollbacks until a consistent global checkpoint is found. As a result a lot of useful computation may be lost and recovery overhead increases.

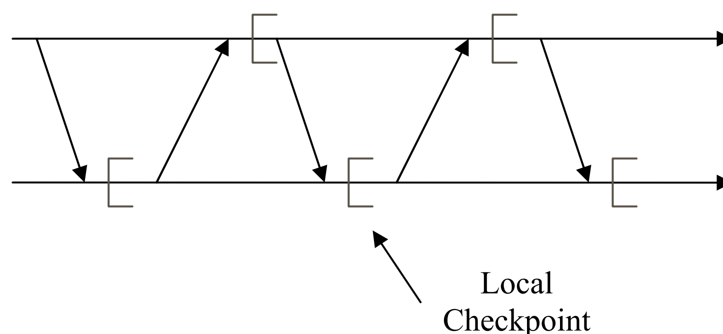


Fig 2.6: Domino Effect

Many of the local checkpoints taken may not be part of any recovery line, and they are called *useless* checkpoints. However, processes may need to store all the local checkpoints since identifying which checkpoints are useless at runtime can be costly. As a result storage requirement is high for this kind of protocols. There is also a possibility of domino effect which may cause a loss of large amount of useful computation [24]. Fig 2.6 shows a

checkpoint and communication pattern involving two processes, which can be affected by domino effect. It can be seen from the figure that all the possible global checkpoints are inconsistent and therefore all local checkpoints are useless checkpoints. If a fault occurs, the recovering process will force the other process to roll back to its previous checkpoint, until both the processes are rolled back to their initial state. In order to determine a recovery line during recovery, processes record their dependencies with checkpoints of other processes during failure-free execution.

The straightforward way to keep track of such information is by using a set of message counters, one for each of the processes with which it communicates. When a process sends out an application message, it increments the counter value corresponding to the receiver process, and tags the value as an identification number to the message. The processes also maintain records of the highest numbered message received from each of its senders. Processes store both these send and receive information along with its checkpoints and this is used to determine inconsistent checkpoints during recovery. A checkpoint of a process P is inconsistent with that of a process Q if the highest message id received from the process Q recorded in P's checkpoint is higher than the send counter value corresponding to P as recorded in Q's checkpoint. Then process P has to roll back again. This may lead to several rounds of coordination and cascaded rollbacks until a consistent global checkpoint is found. These dependency tracking protocols add some overhead during fault-free execution. In case of a fault, all the processes have to coordinate among themselves to decide upon a recovery line to which they will roll back. Therefore, recovery overhead is high in most cases. This approach is very reasonable if faults are rare in the system under consideration. There is also a possibility of domino effect during recovery. This class of protocols does not inherently support output commit.

2.5.2 Synchronous Checkpointing Protocols

In this class of protocols a process does not take local checkpoints independently, but synchronizes every checkpointing event with that of other processes, such that every checkpointing effort results in a consistent global checkpoint. These protocols are also known as synchronous checkpointing protocols. This class of protocols ensures that whenever a process takes a local checkpoint, all other processes in the system also take their respective local checkpoints. As a result every local checkpointing effort translates into a global checkpointing activity. They are free from domino effect. Storage space requirement is

minimum for these protocols, since they require that only the latest checkpoint be stored for recovery. All the checkpoints taken with successful coordination are useful, i.e., the recovery line steadily progresses with every checkpoint. All the latest local checkpoints are part of a consistent global checkpoint, and therefore recovery time is lower compared to asynchronous protocols. However, due to the effort of synchronization involved in every checkpointing activity, checkpointing overhead is high. Synchronous checkpointing protocols can be either blocking or non-blocking.

- Blocking protocols

  A straightforward approach to synchronous checkpointing is to block inter-process communication until the checkpointing protocol completes [9, 32]. The protocol is initiated by a coordinator. The coordinator sends a request to all processes asking them to checkpoint. On the receipt of the request the process blocks normal execution, takes a tentative checkpoint, and sends an acknowledgment message to the coordinator. On receipt of the acknowledgment messages from all the processes, the coordinator sends a message indicating the end of the protocol. On receipt of this message, all processes make their tentative checkpoints permanent, remove old permanent checkpoints, and resume normal execution. Acharya and Badrinath [1] have devised a blocking synchronous checkpointing protocol on the assumption of causal order message delivery by the underlying communication system. Koo and Toueg [18] proposed a blocking synchronous checkpointing protocol which allows failure during checkpointing while storing only two local checkpoints per process. They showed that storage of two checkpoints is the minimal requirement to tolerate failure during checkpointing. Another important feature of their protocol is that only a subset of the processes needs to take checkpoint.

- Non blocking protocols

  The problem with blocking synchronous checkpointing protocols is that the processes are not allowed to execute until the coordination is complete. Hence checkpointing overhead is high. Chandy and Lamport's [8] distributed snapshot protocol provides a non-blocking synchronous checkpointing protocol over reliable FIFO links. They use the property of the underlying link to achieve synchronous checkpointing without the explicit two-round coordination protocol. The idea is to use a special message, called a *marker message*, which is a checkpoint request message carrying the checkpointing interval number. The protocol is similar to the flooding protocols used in distributed systems.. The process which first initiates coordination blocks the local process, takes

a local checkpoint, sends a marker message to all its neighboring processes, and then unblocks the local process. The receiver of a marker message, if has not already received a similar message, follows the same procedure as that of the coordinator and forwards the marker message to all its neighbors. Lai and Yang [19] relaxed the FIFO constraint by piggybacking the marker on every post-checkpoint message. The same affect can be obtained by marking local checkpoints by a sequence number, called checkpoint index, and piggybacking the current checkpoint index value on every application message [12, 28].

Similar to blocking synchronous checkpointing protocols, attempts have been made to construct non-blocking synchronous checkpointing protocols which require that a minimal number of processes take checkpoints. The coordination protocols discussed above involve all the processes in the system and therefore raise concern for scalability of the protocols. Prakash and Singhal [23] observed that only those processes which have communicated since the last checkpoint require to checkpoint again. A minimal number of processes, only those whose present states are causally dependent on the current state of the coordinator, need to participate. They proposed a non-blocking synchronous checkpointing protocol where every process keeps track of the processes which are causally dependent on its present state and the protocol involves only those processes which are causally related to the coordinator.

However, Cao and Singhal [6] found a flaw in Prakash and Singhal's protocol and went on to prove the impossibility of a non-blocking synchronous protocol where a minimal number of processes participate. Cao and Singhal also proposed a non-blocking synchronous checkpointing protocol [7], using mutable checkpoints, which reduces the number of checkpoints to be stored in stable storage. Mutable checkpoints can be stored in the volatile memory and are converted into permanent checkpoints and stored in stable store only when a new recovery line is developed.

Another approach to non-blocking synchronous checkpointing protocol is to avoid explicit coordination by message passing and use synchronous clocks to achieve implicit coordination. In modern distributed systems many applications require clocks of the processors to be approximately synchronized. Many distributed systems run clock synchronization protocol at the background to keep their clock differences within some guaranteed bound [21]. Such loosely coupled synchronized clocks can

facilitate checkpointing effort without explicit coordination .A process takes a local checkpoint and blocks all receives for a period which is equal to the maximum deviation between clocks plus the maximum time to detect a failure in the system. It can be shown that all its local checkpoints of processes form a global recovery line.

2.5.3 Quasi-Synchronous Checkpointing Protocols

This class comprises of protocols which try to combine the advantages of both asynchronous and synchronous checkpointing protocols. This kind of protocols takes two types of checkpoints, namely basic and forced checkpoints. Basic checkpoints can be taken independently by the processes, while forced checkpoints are extra checkpoints forced by the communication pattern to avoid the creation of useless checkpoints in other processes. These protocols ensure that every checkpoint taken by processes is in some recovery line. Since local checkpoints are taken independently these protocols suffer less overhead for checkpointing, and yet avoid domino effect. But these protocols may end up taking more number of checkpoints compared to that in asynchronous checkpointing protocols. The failed process can determine the recovery line without any coordination with other processes. Other processes need to roll back to the recovery line sent by the failed process. Therefore recovery is simple.

The protocols in this class piggyback protocol-specific information on every application messages. The receiver process then analyzes this information to decide whether any forced checkpoint is required or not. If so, the process first takes a checkpoint and then delivers the message to the application. Informally, the decision is based on whether the checkpoint and communication pattern will create any useless checkpoints in the system. If there is such a possibility, a forced checkpoint is taken to break the pattern. The decision is based on the notion of a *Z – cycle* and a *Z – path*, based on the *zigzag path* formulation [8]. A *Z – path* is the same as a zigzag-path. A Z –cycle is a Z – path that begins and ends in the same checkpoint interval (Fig 2.7). CIC protocols can be broadly sub-divided into two classes, *index–based coordination protocols* and *model–based checkpointing protocols*.

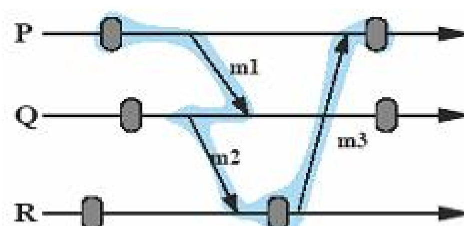


Fig 2.7: *Z*-path determination

In index-based coordination checkpointing protocol, a process takes both basic and forced checkpoints, and all local checkpoints of a process are indexed by a monotonically increasing value. The index value is piggybacked on all application messages. When a process receives an application message, it checks whether the piggybacked index value is higher than its own. If so, then it updates its own index value to the piggybacked index value and takes a forced checkpoint. It then delivers the message to the application process. The protocol ensures that local checkpoints in different processes having the same index value form a recovery line. A more sophisticated protocol where processes transmit more information on application messages to reduce the number of forced checkpoints was proposed in [14].

A model-based checkpointing protocol defines a model of checkpoint and communication pattern which contains no useless checkpoints. For example, a model of checkpoint and communication pattern can be defined as the one which does not have any Z–cycle. A checkpoint and communication pattern, free of Z–cycle, does not contain any useless checkpoints. Therefore, a protocol which enforces such a model ensures that no useless checkpoint is generated in the system. When a process detects a possibility of violation of any constraints put forward by the model, it takes a forced checkpoint. All decisions are taken locally with the help of the information gathered from other processes through piggybacked values. Since the processes take independent decisions, the possibility of violation of the model can be detected by multiple processes at the same time and all of them may take preventive actions. Therefore, processes may end up taking more checkpoints than actually required. Wang has classified these protocols as follows [35].

• *NRAS* or *No – Receive – After – Sendprotocol*, where a checkpoint has to be taken before the first message received after a message has been sent.
• *CAS* or *Checkpoint – After – Send*, where a checkpoint is forced after every send event.
• *CBR* or *Checkpoint – Before – Receive*, where a checkpoint is forced before all receive events.
• *CASBR* or *Checkpoint – After – Send and Before – Receive* is a combination of CAS and CBR protocols. Here, checkpoints are taken after a message is sent and also before a message is received.
• *FDAS* or *Fixed – Dependency – After – Send* uses checkpoint sequence number (*csn*) to track dependency among processes. Every message sent carries this number. If a received

message carries a *csn* value higher than the local *csn* value of the receiver, the receiver process first updates its local *csn* value to the received *csn* value, then takes a forced checkpoint, and then delivers the message to the application process.

- *FDI* or *Fixed Dependency Interval* protocols force checkpoints before dependency of a process changes due to some receive event. Manivannan and Singhal have proposed such a protocol [22].

Helary et. al. showed that since the checkpoint index number in index-based coordination protocols always increases along a Z–path, no Z–cycle can be formed. They have also proved that index-based checkpointing is a form of model-based checkpointing, specifically belonging to the FDAS class.

2.5.4 Log based Recovery Protocol

The previous three classes contain protocols with purely checkpoint-based recovery. Log-based recovery adopts a different strategy for recovery than the earlier three classes. These protocols log all application messages. When a failed process is restarted, the logged messages are replayed to it exactly at the same instances as they were received before its failure. The PWD model ensures that the same execution path is retraced by the process in this technique, i.e., the process can be rolled forward. The execution of a process is modeled as a series of deterministic execution intervals terminated by non-deterministic events, for example, the receive of a message. The first deterministic execution interval of a process begins with the start of the process. Every deterministic execution interval terminates with the first non-deterministic event since its initiation. The same non-deterministic event starts the next deterministic execution interval. If the non-deterministic event *e* terminates an interval *I* and initiates an interval *J*, and if the event *e* can be replayed exactly at the end of interval *I*, then the execution of the interval *J* can be repeated. Recovery in this class of protocols depends on this principle.

These protocols require that the content of the application messages and the information to replay them in order be stored and available during recovery. During recovery the failed process is restarted and all the application messages received by it before failure is replayed. By the PWD model, the failed process can retrace the execution path and will eventually arrive at a state just before the failure, i.e., the process is rolled forward. To avoid a complete restart of the failed process, checkpointing is used in conjunction with message logging. The failed process can then resume execution from a checkpointed state instead of a complete restart. This bounds the amount of rollback a process may suffer and recovery becomes

faster. But the overhead of message logging during failure-free execution is a drawback for these protocols. Checkpointing is used in conjunction with log-based recovery techniques to bound the amount of rollback. Three classes of message log-based recovery protocols have been proposed based on three types of message logging techniques. The problem of output commit is inherently handled by protocols in this class.

- **Pessimistic Message Logging Protocols**

  This kind of protocols takes a conservative view of failure in that a fault can occur immediately after a nondeterministic event. Therefore, they ensure that all non-deterministic events are logged before it can affect the state of a process. All receive messages are logged in the stable storage before it is delivered to the application. Therefore this technique is also referred to as *synchronous logging*. The process which logs a message gets blocked until the message is logged in some stable storage. In this class of protocols, only the failed process needs to recover, and the recovering process can recover up to the last state before failure. No orphan message is created. These classes of protocols always support output commit since the whole history of messages are always logged in the stable storage. Since only the failed process requires to recover, asynchronous checkpointing protocols can be used in conjunction with this class of protocols. So, a process can choose any checkpointing frequency to limit its rollback. Garbage collection is simple because all checkpoints and message log of events prior to the most recent checkpoint can be purged. The overhead of synchronous logging can be reduced by special hardware or by bounding the number of tolerable failures.

- **Optimistic Message Logging Protocols**

  Unlike pessimistic message logging protocols, protocols in this class log messages in the volatile memory and therefore message logging overhead is lower. Message logs in volatile memory are asynchronously flushed to stable storage periodically, without blocking the application processes [29, 30, 36]. This class of protocols optimistically assumes that failure will not occur until the volatile log is flushed into the stable store which allows the failed process to recover to the last state before its failure similar to pessimistic protocols.

However, when a process fails, all message logs stored in its volatile memory are lost and the process can not roll forward to its latest state before failure. If the recovering process had sent messages in the interval between its recovered state and the state before its failure, then such messages become orphan messages. Therefore, the recovering process may create orphan processes in the system.

Consider a situation depicted in Fig 2.8. Let the message m_i be logged by the process p in its volatile log. The process p fails after sending message m_2 and rolls back to C_f. Since m_i is not recorded, the process cannot roll forward to the state before it receives m_i. So m_i becomes an orphan message and q becomes an orphan process. The possibility of creation of orphan processes complicates the recovery and garbage collection processes. Also, processes must be blocked in order to ensure that all volatile message logs are flushed to stable storage to ensure output commit.

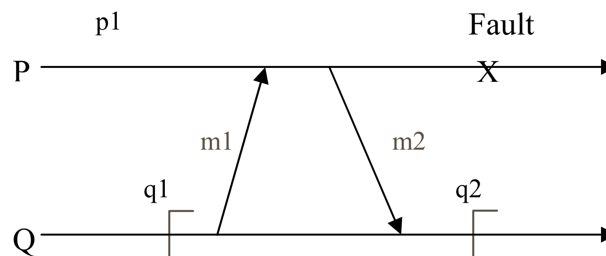


Figure 2.8: Orphan process

To perform rollbacks correctly, optimistic logging protocols track causal dependency during failure-free run. During recovery, this information is used to calculate a global state with no orphan processes. The recovery process is similar to that of asynchronous checkpointing protocols. Multiple processes may need to roll back to avoid the creation of an orphan process. Processes need to store multiple checkpoints. In Fig 2.8 the process q is forced to roll back to C_f in order to bring the system to a consistent state. Moreover, since at failure multiple processes may need to roll back, output commit generally requires the coordination of multiple processes. Recovery can be either *synchronous* or *asynchronous*.

–Synchronous Recovery

In this kind of protocols every process maintains information about causal dependency between processes, developed due to application message transmissions. This information is

then used by the recovering process to determine the processes which need to roll back to avoid the creation of orphan processes.

−***Asynchronous Recovery***

In contrast to synchronous recovery, asynchronous recovery requires that processes participate in multiple rounds of coordination to identify the processes which need to recover and a global state where they can roll back [30]. The process is similar to the recovery procedure for asynchronous checkpointing protocol. It was shown that an exponential number of messages may be exchanged to determine such a global state. The advantage is that, unlike synchronous recovery protocols, these protocols use very small piggybacked information and need not do elaborate book-keeping of dependency information, and as a result have less failure-free overhead. But recovery may become costly.

- **Causal Message Logging Protocols**

  This class of protocols uses the application communication pattern to ensure that whenever a process p sends a message *m*, *p* is recoverable at least up to a state beyond the send event, so that even if *p* fails, the receiver of *m* needs not rollback e_m^p, so that even if *p* fails, the receiver of *m* need not roll back. The protocol satisfies this condition by ensuring that the messages required to roll forward *p* beyond the state e_m^p are either stored in stable storage or are available in the volatile memory of some non-failed processes are either stored in stable storage or are available in the volatile memory of some non-failed processes. Casual logging has the advantage of the low failure-free overhead of optimistic logging, while retaining most of the advantages of the recovery of pessimistic logging [2, 10]. However, recovery and garbage collection procedures are complex.

Chapter 3

INDEX-BASED CHECKPOINTING PROTOCOLS

BCS

Lazy-BCS-Aftersend

BQF

3. INDEX - BASED CHECKPOINTING PROTOCOLS

Index-based checkpointing is a quasi-synchronous approach that allows simple and efficient algorithms for rollback recovery and garbage collection. Checkpoints are time stamped with indexes that are similar to Lamport's logical clocks [20] in a way that checkpoints with the same index form a consistent global checkpoint. The first algorithm to use this approach was the one proposed by Briatico, Ciuffoletti, and Simoncini (BCS) [5]. BCS is very simple and efficient. However, its performance is strongly coupled to the policy adopted to take basic checkpoints. For example, if basic checkpoints are taken periodically according to a global clock, no forced checkpoint is ever taken. In contrast, if each process takes basic checkpoints at different rates, many forced checkpoints may be required. In order to overcome this weakness, many optimizations of BCS were proposed.

The essence of index-based checkpointing is that checkpoints with the same index form a consistent global checkpoint. The goal is to produce an index-based checkpointing protocol that guarantees this property, with a minimum number of forced checkpoints.

3.1 MODEL OF THE DISTRIBUTED COMPUTATION

We consider a distributed computation consisting of n processes $\{P_1, P_2, \ldots, P_n\}$ which interact by message passing. Each pair of processes is connected by a two-way reliable channel whose transmission delay is unpredictable but finite.

Processes are *autonomous* in the sense that they do not share memory, do not share a common clock value, and do not have access to private information of other processes, such as clock drift, clock granularity, clock precision, and speed. Moreover, processes are *heterogeneous* in the sense that private information of the same type of distinct processes is not correlated. We assume, finally, processes follow a fail stop behavior [26].

A process produces a sequence of events and the hth event in process P_i is denoted as $e_{i,h}$; each event moves the process from one state to another. We assume events are produced by the execution of internal, send or receive statements.

The send and receive events of a message m are denoted respectively with *send*(m) and *receive*(m). A *distributed execution* $\hat{E}$ can be modeled as a partial order of events $\hat{E}$ = (E, $\rightarrow$), where E is the set of all events and $\rightarrow$ is the *happened-before relation* [20] defined as follows:

DEFINITION 4.1 *An event $e_{i,h}$ precedes an event $e_{j,k}$, denoted $e_{i,h} \rightarrow e_{j,k}$* iff:

- $i = j$ *and* $k = h + 1$, *or*
- $e_{i,h} = send(m)$ *and* $e_{j,k} = receive(m)$, *or*
- $\exists\ e_{i,z} : (e_{i,h} \rightarrow e_{i,z}) \wedge (e_{i,z} \rightarrow e_{j,k})$.

A checkpoint C dumps the current process state onto stable storage. A checkpoint of process P_i is denoted as $C_{i,sn}$, where sn is called the *index*, or *sequence number*, of a checkpoint. Each process takes checkpoints either at its own pace (*basic* checkpoints) or induced by some communication pattern (*forced* checkpoints). We assume that each process P_i takes an initial basic checkpoint $C_{i,0}$ and that, for the sake of simplicity, basic checkpoints are taken by a periodic algorithm. We use the notation $next(C_{i,sn})$ to indicate the successive checkpoint, taken by P_i, after $C_{i,sn}$. A checkpoint interval $I_{i,sn}$ is the set of events between $C_{i,sn}$ and $next(C_{i,sn})$. Checkpoints are ordered by a relation of precedence, denoted $\rightarrow_C$, and defined as follows:

DEFINITION 4.2. *A checkpoint $C_{i,h}$ precedes a checkpoint $C_{j,k}$, denoted $C_{i,h} \rightarrow_C C_{j,k}$, iff*:

$\exists\, e_{i,l} \in I_{i,g}, \exists\, e_{j,m} \in I_{j,a} : (g \geq h) \wedge (a < k) \wedge (e_{i,l} \rightarrow e_{j,m})$.

More simply, a checkpoint $C_{i,h}$ precedes a checkpoint $C_{j,k}$ if there is causal path of messages starting after $C_{i,h}$ and ending before $C_{j,k}$.

A global checkpoint C is a set of local checkpoints $\{C_{1,sn_1}, C_{2,sn_2}, \ldots C_{n,sn_n}\}$ one from each process.

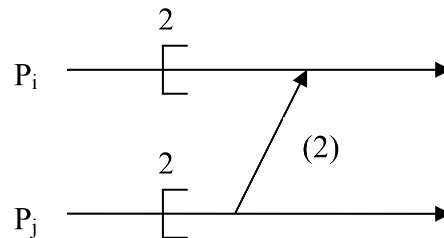
DEFINITION 4.3 A global checkpoint $\{C_{1,sn_1}, C_{2,sn_2}, \ldots C_{n,sn_n}\}$ is consistent iff

$$\forall\, i, j \in [1, n]: i \neq j \Rightarrow \neg\left(C_{i,sn_i} \rightarrow_C C_{j,sn_j}\right).$$

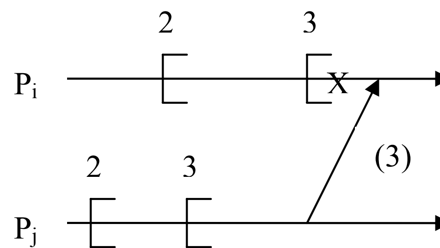
In the following, we denote with C_{sn} a global checkpoint formed by checkpoints with sequence number sn and use the term consistent global checkpoint C_{sn} and recovery line $\mathcal{L}_{sn}$ interchangeably.

3.2 BCS

In the algorithm proposed by Briatico, Ciuffoletti, and Simoncini (BCS) [5], every process maintains and propagates an index s_n that is similar to a logical clock [20]. Process P_i initializes *sn* to 0 and increments it after a basic checkpoint is taken. When P_i sends a message, it piggybacks *sn* onto it. When P_i receives a message with $m.sn > sn$ it takes a forced checkpoint as shown in Fig 3.1



(a) P_i does not increase its index



(b) P_i must increment its index

Fig 3.1 BCS

The Briatico-Ciuffoletti-Simoncini (BCS) algorithm can be sketched by using the following rules associated with the action to take a local checkpoint:

take-basic (BCS):

When a basic checkpoint is scheduled

$sn_i \leftarrow sn_i + 1$

a checkpoint $C_{i, sn}$ is taken

take-forced (BCS):

Upon the receipt of message *m***:**

if $sn_i < m.sn_i$ **then** a checkpoint $C_{i,m.sn}$ is taken

By using the above rules, it has been proved that C_{sn} is consistent [5]. Note that, due to the rule **take- forced (BCS)**, there could be some gap in the index assigned to checkpoints by a process. Hence, if a process has not assigned the index *sn,* the first local checkpoint of the

process with sequence number greater than *sn* can be included in the consistent global checkpoint C_{sn}.

Each time a basic checkpoint is taken, *sn* is incremented by one and the process starts a lazy coordination to form a consistent global checkpoint C_{sn}. In the worst case of BCS algorithm, the number of forced checkpoints induced by a basic one is n -1. In the best case, if all processes take a basic checkpoint at the same physical time, the number of forced checkpoints per basic one is zero. However, in an *autonomous environment,* local indices of processes may diverge due to many causes (process speed, different period of the basic checkpoint etc). This pushes the indices of some processes higher and each time one of such processes sends a message to another one, it is extremely likely that a number of forced checkpoints, close to n - 1, will be induced

3.3 LAZY-BCS-AFTERSEND

Let us consider a checkpoint interval in which a process P_i has only received messages with indexes that are smaller than its current index. Process P_i can deduce that it is ahead and not increment its index when the next checkpoint is taken. On the contrary, if P_i has received at least one message with the same index, P_i must increment its index when it takes the next basic checkpoint. In order to implement this behavior a process needs to maintain a flag that indicates whether a message with an equal index has arrived in the current checkpoint interval. This optimization, called *Lazy-BCS*, reduces the impact of asymmetry and guarantees the absence of the domino effect [4, 14].

Let us consider an interval in which a process P_i has not sent any message at the time it receives a message with a greater index. Since has not propagated the index of the current interval, it can increase its index without taking a forced checkpoint. We call this approach *aftersend* and it is domino-effect free. In order to implement this behavior a process needs to maintain a flag that indicates whether a message has been sent. This optimization has been incorporated to many index-based protocols and has also appeared in the context of checkpointing protocols that enforce Rollback-Dependency Tractability [35]. By combining the previous two optimizations we get Lazy-BCS Aftersend.

The Briatico-Ciuffoletti-Simoncini (BCS) algorithm can be sketched by using the following rules associated with the action to take a local checkpoint:

take-basic (Lazy-BCS Aftersend):

When a basic checkpoint is scheduled:

if $skip_i$ **then** $skip_i \leftarrow$ False

else $sn_i \leftarrow sn_i + 1$;

a checkpoint $C_{i, sn}$ is taken;

take-forced (Lazy-BCS Aftersend):

Upon the receipt of a message m:

if $sn_i < m.sn$ **then**

if *aftersend* then $sn_i \leftarrow m.sn$

else $sn_i \leftarrow m.sn$

a checkpoint $C_{i,m.sn}$ is taken

the message is processed;

3.4 BQF

The algorithm proposed by Baldoni, Quaglia and Fornara (BQF) [4], is an index based checkpointing algorithm in which each process maintains an index *sn* and updates the sequence number *sn* as a checkpoint is taken. This points out that forced checkpoints are due to the process of fast increasing of the sequence numbers. So, in order to slow down this phenomenon, we define an equivalence relation between successive checkpoints of a process. This relation allows a recovery line to advance without increasing its sequence number. From an operational point of view, the equivalence between checkpoints can be detected by a process exploiting causal dependencies between checkpoints. This algorithm embeds such a mechanism to detect equivalences between checkpoints by using a vector of integers piggybacked on application messages. In the worst case, this algorithm takes the same number of checkpoints as the algorithm in [22].The price paid is that each application message piggybacks more controls information (one vector of messages) compared to the previous proposals.

3.4.1 Equivalence Between Checkpoints

DEFINITION 4.4. Two local checkpoints $C_{i,sn}$ and *next*($C_{i,\ sn}$) of process P_i are equivalent with respect to the recovery line $\mathcal{L}_{sn}$ (including $C_{i,sn}$), denoted $C_{i,sn} \overset{L_{sn}}{\equiv} next(C_{i\ sn})$, if

$$\forall\ C_{j,\ sn} \in \mathcal{L}_{sn}\text{: } j \neq i\ \neg\left(C_{j,sn} \rightarrow_C next(C_{i,sn})\right).$$

As an example, consider the recovery line $\mathcal{L}_{sn}$ depicted in Fig. 3.2a, where checkpoints are depicted by thick crosses and arrows between processes represent messages. If in $I_{2,sn}$ process P_2 executes either send events or receive events of messages which have been sent from the left side of $\mathcal{L}_{sn}$, then $C_{2,sn} \overset{L_{sn}}{\equiv} next(C_{2,sn})$ and a recovery line $\mathcal{L}`_{sn}$ can be created by replacing $C_{2,sn}$ with *next*($C_{2,sn}$) from $\mathcal{L}_{sn}$. Fig. 3.2b shows the construction of the recovery line $\mathcal{L}``_{sn}$ starting from $\mathcal{L}`_{sn}$ by using the equivalence between $C_{1,sn}$ and *next*($C_{1,sn}$) with respect to $\mathcal{L}`_{sn}$. Hence, we can say, $C_{i,sn}$ is not equivalent to *next*($C_{i,sn}$) with respect to $\mathcal{L}_{sn}$ if at least one message is sent from the right side of $\mathcal{L}_{sn}$ and is received by P_i in $I_{i,sn}$.

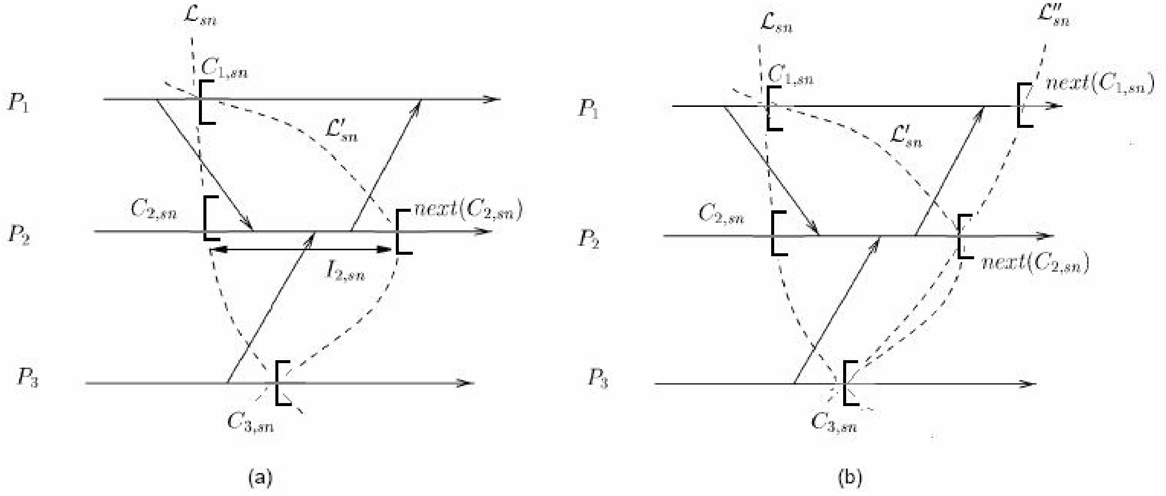


Fig 3.2 Examples of pairs of equivalent checkpoints

From the above examples, a simple property follows:

PROPERTY 4.1 If $C_{i,sn} \overset{L_{sn}}{\equiv} next(C_{i,sn})$ then $\mathcal{L}`_{sn}$ then $\mathcal{L}`_{sn} = \mathcal{L}_{sn} - \{C_{i,sn}\} \cup \{next(C_{i,sn})\}$ is a recovery line.

PROOF:

$C_{i,sn} \overset{L_{sn}}{\equiv} next(C_{i,sn})$

As $\mathcal{L}_{sn}$ is a recovery line including $C_{i,sn}$ then

$\forall\ C_{j,sn} \in \mathcal{L}_{sn}\text{: } j \neq i\ \neg\left(C_{j,sn} \rightarrow_C next(C_{i,sn})\right).$

So, the set of local checkpoints $\mathcal{L}_{sn} - \{C_{i,sn}\} \cup \{next(C_{i,sn})\}$ is a consistent set.

Hence, if a process detects a pair of equivalent checkpoints, it can advance the recovery line without updating its sequence number.

3.4.2 Tracking the Equivalence Relation On-The-Fly

The events influencing the detection of the equivalence are: the arrival of a message (which enlarge the knowledge about the causal past of a process) and the event of taking a basic checkpoint.

Upon the arrival of a message *m* at P_i in the checkpoint interval $I_{i,sn,en}$: One of the following three cases is true:

1) ($m.sn < sn_i$) or (($m.sn = sn_i$) and ($\forall$ $m.EQ[j] < EQ_i[j]$)):

% *m* has been sent from the left side of the recovery line $\bigcup_{\forall j} C_{j,sn,EQ_i[j]}$ %

2) ($m.sn = sn_i$) and ($\exists j$: $m.EQ[j] \geq EQ_i[j]$):

% *m* has been sent from the right side of the recovery line $\bigcup_{\forall j} C_{j,sn,EQ_i[j]}$%

3) ($m.sn > sn_i$)

% *m* has been sent from the right side of a recovery line of which P_i was not aware %

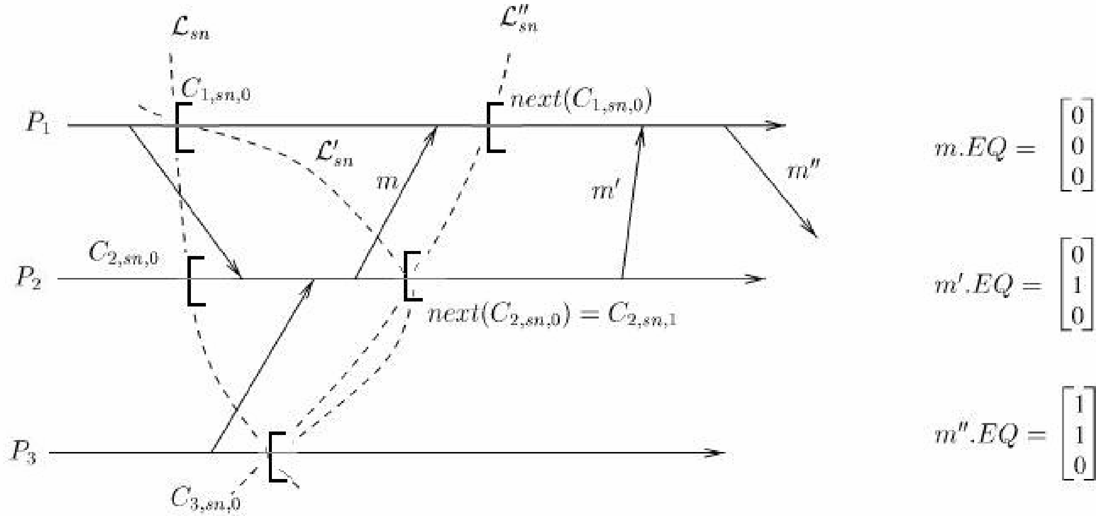


Fig 3.3 Upon the receipt of m`, P_1 detects $C_{1,sn,0} \overset{L_{sn}}{\equiv} next(C_{1,sn,0})$

A message *m* which falls in case **3)** directs P_i to take a forced checkpoint $C_{i,m.sn\ 0}$. So, the only interesting cases for tracking the equivalence are **1)** and **2)**.

At the time of the basic checkpoint $next(C_{i,sn,en})$**:** P_i falls in one of the following two alternatives:

1) If no message is received in $I_{i,sn,en}$ that falls in case **2)**, then $C_{i,sn,en} \overset{L_{sn}}{\equiv} next(C_{i,sn,en})$. That equivalence can be tracked by a process using its local context at the time of the checkpoint $next(C_{i,sn,en})$. Thus, $next(C_{i,sn,en}) = C_{i,sn,en+1}$ (the equivalence between, $C_{2,sn,0} \overset{L_{sn}}{\equiv} next(C_{2,sn,0})$ shown in Fig. 3.3, is an example of such a behavior)

2) If there exists at least one message m received in $I_{i,\, sn,\, en}$ which falls in case 2), one checkpoint belonging to the recovery line $\bigcup_{\forall j} C_{j,sn,EQ_i[j]}$ precedes $next(C_{i,sn,en})$ this communication pattern is shown in Fig.3.3,where $\bigcup_{\forall j} C_{j,sn,EQ_i[j]}=\{C_{1,sn,0},C_{2,sn,0},C_{3,sn,0}\}$ and due to m, $C_{2,sn,0} \rightarrow_C next(C_{1,sn,0})$. The consequence is that process P_i cannot determine, at the time of taking the basic checkpoint $next(C_{i,sn\, en})$ if $C_{i,sn,en}$ is equivalent to $next\ (C_{i,\, sn,\, en})$ with respect to some recovery line. As an example, in Fig 3.3, process P_1 cannot determine if $C_{1,sn,0}$ is equivalent to $next(C_{1,sn,0})$ with respect to some recovery line when taking $next(C_{1,sn,0})$.

To solve the problem raised in point **2)**, two approaches can be pursued. If, at the time of the basic checkpoint $next(C_{i,sn,en})$, the equivalence between $C_{i,sn,en}$ and $next(C_{i,sn,en})$ is undetermined, then:

Pessimistic Approach: Process P_i assumes pessimistically $next(C_{i,sn,en}) = C_{i,sn+1,0}$ even though this determination could be revealed wrong in the future of the computation. Fig. 3.3 shows a case in which message m\` brings the information (encoded in $m$\`$.EQ$) to P_1 that $C_{2,sn,0} \overset{L_{sn}}{\equiv} next(C_{2,sn,0})$ and that the recovery line was advanced by P_2, from $\mathcal{L}_{sn}$ to $\mathcal{L}$\`$_{sn}$. In such a case, $P_1$ can determine $C_{1,sn,0}$ is equivalent to $next(C_{1,sn,0})$ with respect to $\mathcal{L}$\`$_{sn}$.

Optimistic Approach: Process P_i assumes *optimistically* (and *provisionally*) that $C_{i,sn,en}$ is equivalent to $next(C_{i,sn,en})$. So, the index of $next(C_{i,sn,en})$ becomes <*sn*,*en*+1>.As provisional indices cannot be propagated in the system, if *at the time of the first send event after* $next(C_{i,sn,en})$ the equivalence is still undetermined, then $next(C_{i,sn,en}) = C_{i,sn+1,0}$ (thus, sn_i=sn_i+1, $en_i = 0$, and $\forall j$: $EQ_i[j] = 0$). Otherwise, the provisional index becomes permanent. Fig. 3.3 shows a case in which $C_{1,sn,0} \overset{L_{sn}}{\equiv} next(C_{1,sn,0})$ and this is detected by P_i before sending m\`\`. After m\`\` is sent, the index <*sn*, 1> of $next(C_{1,sn,0})$ becomes permanent.

3.4.3 The BQF Algorithm

We assume each process P_i has the following data structures:

sn_i, en_i**: integer;**

$after_first_send_i$, $skip_i$, $provisional_i$**: Boolean;**

$past_i$, $present_i$, EQ_i**:** ARRAY [1, n] of **integer**.

$present_i[j]$ represents the maximum equivalence number en_j sent by P_j in the current interval, and piggybacked on a message that falls in case 2) of Section 3.4.2. Upon taking a checkpoint or when updating the sequence number, all the entries of $present_i$ are initialized to -1. If the checkpoint is basic, $present_i$ is copied in $past_i$ before its initialization. Each time a message m is received such that $past_i[h] < m.EQ[h]$, $past_i[h]$ is set to -1. So,the predicate ($\exists h$: $past_i[h]$>-1) indicates that there is a message received in the past checkpoint interval that has been sent from the right side of the recovery line *currently* seen by P_i (case 2 of Section 3.4.2).Below, the process behavior is shown (the procedures and the message handler are executed in atomic fashion). This implementation assumes that there exists at most one provisional index in each process. So, each time two successive provisional indices are detected, the first index is permanently replaced with <sn_i + 1, 0>.

init P_i:

$sn_i = 0$;

$en_i = 0$;

$after_first_send_i = FALSE$;

$skip_i = FALSE$;

$provisional_i = FALSE$;

$\forall h\ EQ_i[h] = 0$;

$\forall h\ past_i[h] = -1$;

$\forall h\ present_i[h] = -1$;

When *m* arrives at P_i from P_j:

```
if m.sn > sn_i then          % P_i is not aware of the recovery line with sequence number m.sn %
begin
        if after_first_send_i then
        begin
                take a checkpoint;        % taking a forced checkpoint %
                skip_i = TRUE;
                after_first_send_i = FALSE;
        end;
```

$sn_i = m.sn$; $en_i = 0$;

assign the index $<sn_i, en_i>$ **to the last taken checkpoint**;

$provisional_i = FALSE$; % the index is permanent %

$\forall\ h\ past_i[h] = -1$;

$\forall\ h\ present_i[h] = -1$;

$present_i[j] = m.EQ[j]$;

$\forall\ h\ EQ_i[h] = m.EQ[h]$;

end

else if $m.sn = sn_i$ then

begin

if $present_i[j] < m.EQ[j]$ then $present_i[j] = m.EQ[j]$;

$\forall\ h\ EQ_i[h] = max\ (EQ_i[h], m.EQ[h])$; % a component-wise maximum is performed %

$\forall\ h$ if $past_i[h] < m.EQ[h]$ then $past_i[h] = -1$;

end;

process the message m;

When P_i sends data to P_j:

if $provisional_i \wedge (\exists\ h{:}\ past_i[h] > -1)$ **then** %last checkpoint not equivalent to the previous one%

begin

$sn_i = sn_i + 1$; $en_i = 0$;

assign the index $<sn_i, en_i>$ **to the last taken checkpoint**;

$provisional_i = FALSE$; % the index is permanent %

$\forall\ h\ past_i[h] = -1$;

$\forall\ h\ present_i[h] = -1$;

$\forall\ h\ EQ_i[h] = 0$;

end;

m.content = *data*;

m.sn = *sni*;

$m.EQ = EQ_i$; % packet the message %

send (*m*) **to** *Pj*; *after_first_sendi* = *TRUE*;

When a basic checkpoint is scheduled from *Pi*:

if $skip_i$ then $skip_i = FALSE$ % the basic checkpoint is skipped %

else

begin

if *provisionali* then % two successive provisional indices %

if ($\exists h: past_i[h] > -1$) then % last checkpoint not equivalent to the previous one %

begin

$\forall h \; past_i[h] = -1$;

$sn_i = sn_i + 1$;

$en_i = 0$;

assign the index$<sn_i, en_i>$ **to the last taken checkpoint**;% permanent index%

$\forall h \; EQ_i[h] = 0$;

end

else $\forall h \; past_i[h] = present_i[h]$; % last checkpoint is equivalent to the previous one %

take a checkpoint; % taking a basic checkpoint %

$en_i = en_i + 1$;

$EQ_i[i] = en_i$;

assign the index $<sn_i, en_i>$ **to the last taken checkpoint**;

$provisional_i = TRUE$; % the index is provisional %

$\forall h \; present_i[h] = -1$;

$after_first_send_i = FALSE$;

end

Chapter 4

ENHANCED INDEX-BASED CHECKPOINTING PROTOCOL

Simulation Environment

The Algorithm

Drawbacks of BQF

Experimental Results

4. ENHANCED INDEX BASED CHECKPOINTING ALGORITHM

The algorithm previously described, BCS [5], Lazy-BCS Aftersend [35], BQF [4] start will less complexity and increase in complexity. Among the three BQF is efficient it has less number of forced checkpoints at the cost more control information. In this chapter we propose an algorithm that is more efficient both in terms of number of checkpoints and the control information that is needed to propagate among the processes. The proposed algorithm requires only the sequence number to be piggybacked along with the message instead of the sequence number and EQ_i (equivalence vector) as in case of BQF. The EQ_i needs to be piggybacked in BQF as it is optimistically assuming that the $C_{i,sn,en}$ and $next(C_{i,sn,en})$ are equivalent. This optimistic assumption has made BQF to propagate an additional vector EQ_i each time a message is being transmitted by a process. In this proposed algorithm we haven't made such assumption which obviously decreased at cost at which we have achieved less number of forced checkpoints than the others.

In this algorithm as all the index based checkpointing algorithms each process maintains a sequence number *sn*, which is incremented by one as a process takes a basic checkpoint or a forced checkpoint. This sequence number *sn* is piggybacked along with all the messages that are transmitted from this process. If a process receives a message with a sequence number, *m.sn*, higher than its own sequence number then that process is forced to take a checkpoint with the sequence number of the message, *m.sn*.

Each process is also associated with a boolean variable *inc* which indicates whether to increment or not when a basic checkpoint is taken. This boolean variable *inc* is set to true only if the process receives atleast a message with sequence number, *m.sn* greaterthan or equal to the sequence number, *sn*, of the process. This variable allows the processes to increase the sequence number slowly. This makes process to be in synchronous with the other processes. If all the processes are increasing the sequence number at the same phase then the need to take the forced checkpoints is removed ultimately removing the number of forced checkpoints.

Each process is also associated with another boolean variable, *aftersend,* which indicates whether there is atleast one sending event in the current interval since the last checkpoint. If this variable is not set during the current interval at the time of reception of the message with sequence number, *m.sn*, higher than the sequence of the process, as the aftersend boolean

variable is not set, which indicates that there is no sending event since the last checkpoint there is no need to force the process to take a checkpoint instead the sequence number, $C_{i,sn}$, of the previous checkpoint is updated with the sequence number of the message, *m.sn*, without taking the checkpoint. As the process has not revealed its sequence number, *sn*, in the current interval to the other process the above optimization is handful. On the other hand if this boolean variable, *aftersend*, is set the process is forced to take a checkpoint with the sequence number of the message, *m.sn*.

The checkpoint interval of all the processes is dynamically updated depending on when a process takes a forced checkpoint. As a process takes a forced checkpoint the checkpoint interval of that process is reset and the new interval starts from this moment. Again the length of this interval will be same, that is, an average of 10 communication events from now. This reset strategy has also worked well to keep a process in phase with the remaining other processes in terms of the sequence numbers. The interval is thus determined dynamically as opposed to the static interval implemented by the previous algorithms.

4.1 Simulation Environment

Our experimental data was obtained using SPIN [3] - the simulator and model checker. Promela is the input language of SPIN .In Promela, the processes are asynchronous and the simulated execution of each process is a succession of atomic events of three types: internal, send-message and receive message. The only type of internal event that is relevant for checkpointing is the occurrence of a basic checkpoint. The environment is controlled by adjusting the distribution of these events and the communication network.

All of the experiments were performed considering a complete network, i.e., each pair of processes is connected by a bidirectional communication channel. The channels do not lose or corrupt messages. For each experimental point it was considered the average of 10 measurements. Each measurement was taken by the execution of each of the studied algorithms under the same pattern of messages and basic checkpoints. We counted the ratio of forced checkpoints per total messages with an average of 500 per process

We considered two scenarios in this study [34]:

None-faster: In the none-faster scenario all processes have, on average, the same number of events between basic checkpoints. The processes do not take basic checkpoints at exactly the same time; just the ratio of basic checkpoints per events is the same. This setting represents a situation where all the processes behave in the same way and have the same execution speed. Particularly, we have an average of 10 communication events between any two basic checkpoints. We have made the measurements varying the number of processes from 2 to 15.

One-faster: In the one-faster scenario all but one process behaves as in the none-faster scenario and this process has a smaller number of events between the basic checkpoints. This setting represents a system with a single process that has more important local states and is willing to take more basic checkpoints, introducing asymmetry in the pattern of basic checkpoints. In this scenario we consider a system in which one process will have half the interval than the others; one process is taking the basic checkpoints double the rate at which other processes are taking. Particularly we have considered the first process 5 communication events between any two basic checkpoints and the others with 10 communication events between any two basic checkpoints.

4.2 SPIN

SPIN is a popular open-source software tool, used by thousands of people worldwide that can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original UNIX group of the Computing Sciences Research Center, starting in 1980. The software has been available freely [31] since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.

SPIN is a tool for analyzing the logical consistency of distributed systems, specifically of data communication protocols [16]. The system is described in a modeling language called Promela (Process or Protocol Meta Language). The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered). XSPIN is a graphical front-end to drive SPIN (written in Tcl/Tk) [3, 25].

Given a model system specified in Promela, SPIN can perform random or interactive simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. During simulations and verifications SPIN checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to prove the correctness of system invariants and it can find non-progress execution cycles. Finally, it supports the verification of linear time temporal constraints; either with Promela never-claims or by directly formulating the constraints in temporal logic.

The verifier is setup to be fast and to use a minimal amount of memory. The exhaustive verifications performed by SPIN are conclusive. They establish with certainty whether or not a system's behavior is error-free. Very large verification runs, that can ordinarily not be performed with automated techniques, can be done in SPIN with a ``bit state space'' technique. With this method the state space is collapsed to a few bits per system state stored. Although this technique doesn't guarantee certainty, the coverage is better, and often much better, than that obtained with traditional random simulation.

SPIN is a generic verification system that supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking,

As a formal methods tool, SPIN aims to provide:

1) an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail,

2) a powerful, concise notation for expressing general correctness requirements, and

3) a methodology for establishing the logical consistency of the design choices from 1) and the matching correctness requirements from 2).

Much formalism has been suggested to address the first two items, but rarely are the language choices directly related to a basic feasibility requirement for the third item. In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language PROMELA (a Process Meta Language) [13], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL).

There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in PROMELA are, therefore, always required to be bounded, and have only countably many distinct behaviors. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs. All verification systems, of course, do have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure. These constraints are an often neglected issue in formal verification.

4.2.1 Structure

The basic structure of the SPIN model checker is illustrated in Fig. 4.1 [15]. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN's graphical front-end XSPIN. After fixing syntax errors, interactive simulation is performed until basic confidence is gained that the design behaves as intended. Then, in a third step, SPIN is used to generate an optimized on-the-fly verification program from the high level specification. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If any counterexamples to the correctness claims are detected, these can be fed back into the interactive simulator and inspected in detail to establish, and remove, their cause.

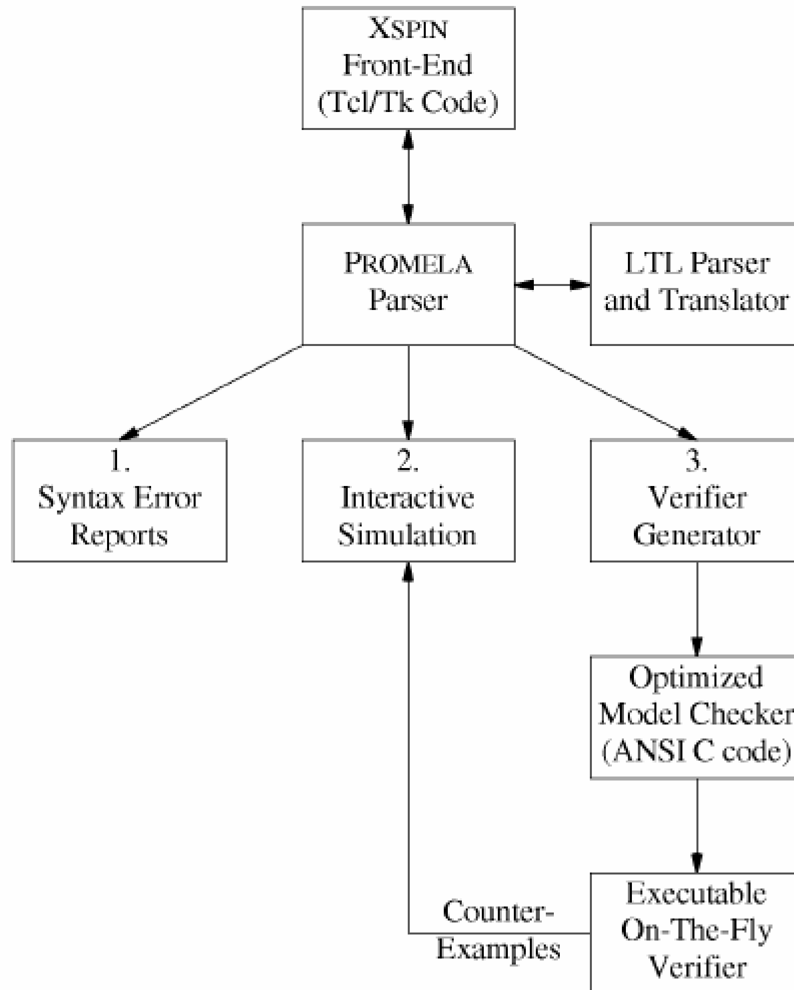


Fig 4.1 The structure of SPIN simulation and verification

4.3 THE ALGORITHM:

init P_i:

$sn_i = 0$;

$aftersend_i = FALSE$;

$inc_i = FALSE$;

$interval_i = 10$;

When *m* arrives at P_i from P_j:

if $m.sn > sn_i$ then % P_i is forced to take a checkpoint with sequence number $m.sn$ %

begin

if $aftersend_i$ then

begin

```
            take a checkpoint m.sn as the index;        % taking a forced checkpoint %
            interval_i =0;
            aftersend_i = FALSE;
            inc_i = TRUE;
            sn_i = m.sn;
      end;
else
      begin
            inc_i = TRUE;                    % a forced checkpoint has been skipped %
            sn_i = m.sn;
            assign the index sn_i to the last checkpoint taken
      end
end
else if m.sn = sn_i then inc_i =TRUE;
process the message m;
```

<u>When P_i sends data to P_j:</u>

```
m.content = data;                            % packet the message %
m.sn = sn_i;
send (m) to P_j;
aftersend_i =TRUE;
interval_i = interval_i – 1;
```

<u>When a basic checkpoint is scheduled from P_i:</u>

```
If inc_i then sn_i = sn_i + 1;
take a checkpoint with index as sni
interval_i = 0; aftersend_i = FALSE; inc_i = FALSE;
```

4.4 DRAWBACKS OF BQF

- As BQF is optimistically assuming that $C_{i,sn,en}$ is equivalent with $next(C_{i,sn,en})$ to implement this assumption it is three vectors EQ_i, $present_i$, $past_i$ which is unnecessarily increasing the complexity of the algorithm.
- Because of the same assumption that $C_{i,sn,en}$ is equivalent with $next(C_{i,sn,en})$ BQF is also increasing the message traffic by increasing the amount of message that is to be piggybacked on each message by the process

The above two drawbacks are overcome in the proposed algorithm by merely not assuming that $C_{i,sn}$ and $next(C_{i,\ sn})$ are equivalent. This ultimately reduced the amount of control message traffic that is to be piggybacked on a message whenever a process needs to send some information to the other process over the network. Also the complexity of the algorithm has been decreased by not needing to maintain unnecessary vector EQ_i, presenti, pasti, as needed by BQF.

The proposed algorithm also follows a new strategy to update the checkpoint interval dynamically as opposed to the static interval used by the existing algorithms explained in the previous chapter. Whenever a process takes a forced checkpoint due to the reception of a message with sequence number higher than the sequence number of the process, the checkpoint interval is reset that is a new interval starts from the point where the forced checkpoint is taken. By using the above rules the set of local checkpoints one from each process with same sequence numbers forms a consistent global checkpoint. It may also happen that some processes may not have a particular sequence numbers in such case the next local checkpoint from that process with the sequence number greater than or equal to the sequence number of the consistent checkpoint is included into the global consistent set.

4.5 EXPERIMENTAL RESULTS

In this section, simulation results are demonstrated to compare the protocol proposed in this thesis to traditional ones. The simulation is built on SPIN, a tool to trace logical design errors and check the logical consistency of protocols and algorithms in distributed systems. The basic checkpoint interval is set to approximately equal to every 10 send events. There are only three types of events, that is, sending events, receiving events and checkpointing events.

The number of forced checkpoint events and the number of computation messages will be counted every 500 send events for each process. Process chooses its destination randomly in sending events. It is assumed that messages will not be corrupted or lost or disordered. The compared parameter is the ratio between the number of forced checkpoint events of a protocol and the number of computation messages [33], which indicates the performance of how many forced checkpoints per computation message. The numbers of processes vary from 2 to 15.

The simulation result for none faster scenario is depicted in Fig.4.2 here the X-axis represents the number of processes and Y-axis represents the ratio between the number of forced checkpoints and the number of messages of execution. The simulation shows that the proposed protocol outperforms than the other three protocols. It reduces the number of forced checkpoints by 60.9% as compared to BCS, 45% as compared to Lazy-BCS-Aftersend and 31.4% as compared with BQF.

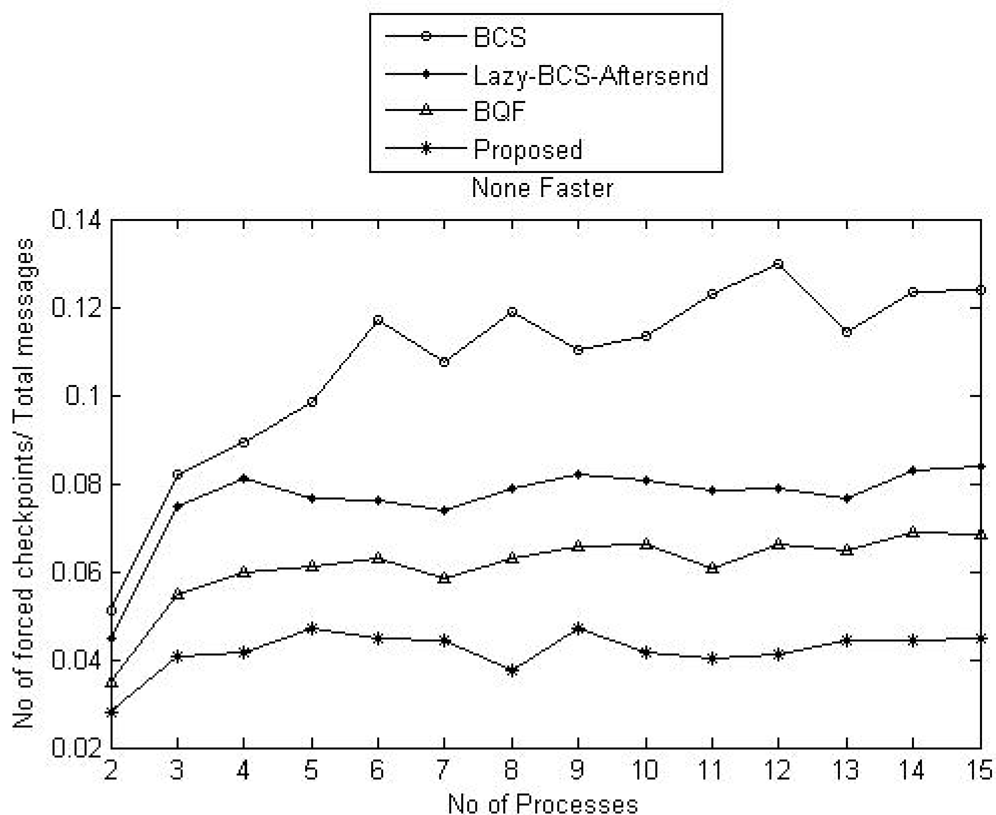


Fig 4.2 None faster situation

The simulation result for one faster scenario is depicted in Fig.4.3.We have considered 10 send events as the interval for all the processes and 5 send events as the interval for the first process. The X-axis represents the number of processes and Y-axis represents the ratio between the number of forced checkpoints and the number of messages of execution. The simulation shows that the proposed protocol outperforms than the other three protocols. It reduces the number of forced checkpoints by 55.1% as compared to BCS, 46.4% as compared to Lazy-BCS-Aftersend and 27.6% as compared with BQF.

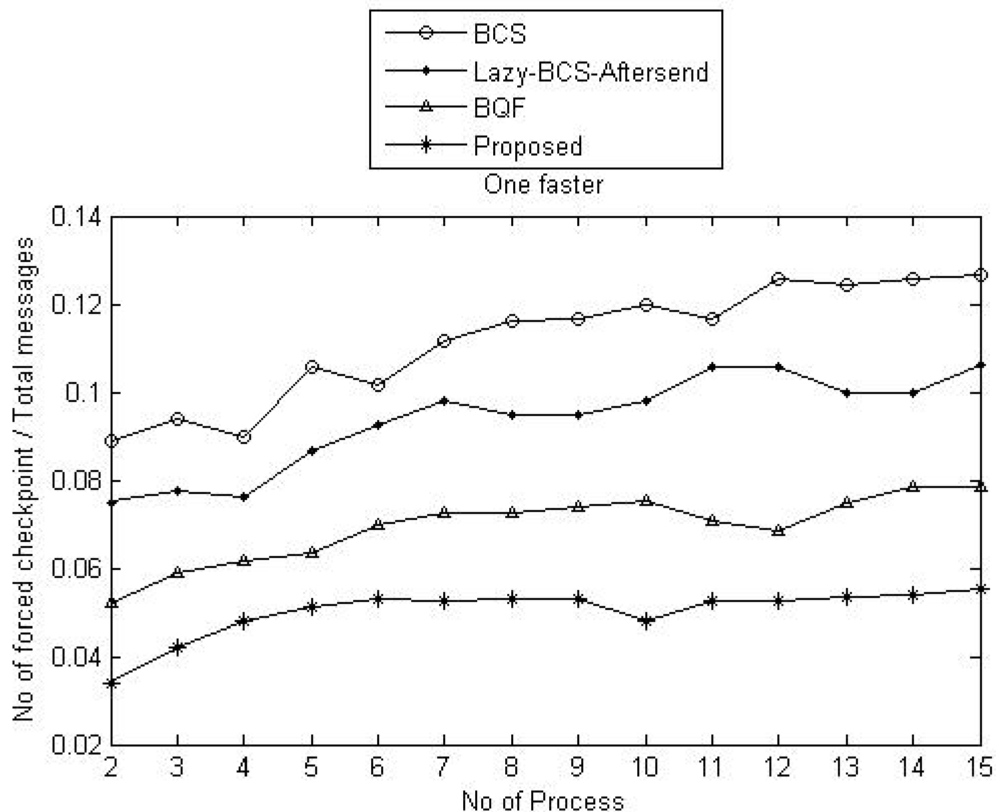


Fig 4.3 One faster situation

Simulation results show that the proposed scheme can reduce the number of forced checkpoints per message about 27-32% on an average when compared to BQF.

Chapter 5

CONCLUSION AND FUTURE WORK

5. CONCLUSION AND FUTURE WORK

This thesis presents an Enhanced index-based Checkpointing algorithm for Distributes Systems to improve the performance of index-based checkpointing algorithms. The main drawbacks of the existing BQF, is that first it optimistically assuming that $C_{i,sn,en}$ is equivalent with $next(C_{i,sn,en})$ to implement this assumption it is three vectors EQ_i, $present_i$, $past_i$ which is unnecessarily increasing the complexity of the algorithm. Secondly, because of the same assumption that $C_{i,sn,en}$ is equivalent with $next(C_{i,sn,en})$, BQF is also increasing the message traffic by increasing the amount of message that is to be piggybacked on each message by the process.

The above two drawbacks are overcome in the proposed algorithm by merely not assuming that $C_{i,sn}$ and $next(C_{i,sn})$ are equivalent. The proposed algorithm also follows a new strategy to update the checkpoint interval dynamically as opposed to the static interval used by the existing algorithms explained in the above sections. Whenever a process takes a forced checkpoint due to the reception of a message with sequence number higher than the sequence number of the process, the checkpoint interval is reset that is a new interval starts from the point where the forced checkpoint is taken. The timer reset strategy of resetting the timer after a forced checkpoint restarts a new basic checkpoint interval.

Thus the Enhanced index-based checkpointing algorithm decreases the number of checkpoints with less complexity as compared to the existing index-based checkpointing algorithms.

FUTURE WORK

The proposed algorithm can be extended to incremental checkpointing wherein instead of saving the complete state of the process only the pages which have been modified by a particular process are saved each time a local checkpoint is taken by a process. This reduces the checkpointing overhead, as only part of a process state is saved instead of saving the complete state.

REFERENCES

[1] ACHARYA A. and BADRINATH B. R., "Recording distributed snapshots based on causal order of message delivery." Information Processing Letters 44, pp. 317-321, 1992.

[2] ALVISI L., "Understanding the message logging paradigm for masking process crashes." Ph.D. dissertation, Cornell University, January 1998.

[3] Basic Spin Manual. http://spinroot.com/spin/Man/Manual.html. 28 November 2003.

[4] BALDONI R., QUALIGIA F., and FORNARA P. "An index-based checkpointing algorithm for autonomous distributed systems." IEEE Trans. Parallel and Distributed Syst. Volume 10 No 2, (February 1999): p. 181-192.

[5] BRIATICO D., CIUFOLETTI A., and SIMONCINI L. "A distributed domino-effect free recovery algorithm." In Proc. 4th IEEE Symp. On Reliability in Distributed Software and Database Syst., (1984): p. 207-215.

[6] CAO G. and SINGHAL M., "On coordinated checkpointing in distributed system." IEEE Trans, on Parallel and Distributed Systems Volume 9, No 12, (December 1998): p. 1213-1225.

[7] CAO G and SINGHAL M., "Mutable checkpoints: A new checkpointing approach for mobile computing systems." IEEE Trans, on Parallel and Distributed Systems Volume 12, No 2, (February 2001): p. 157-172,

[8] CHANDY K. M. and LAMPORT, "Distributed snapshots: Determining global states in distributed systems." ACM Trans on Computer Systems, Volume 3, No. 1, (February 1985): p. 63-75.

[9] CRITCHLOW C and TAYLOR K., "The inhibition spectrum and the achievement of causal consistency", Cornell University." (February 1990) Tech. Rep. TR 90-1101.

[10] ELNOZAHY and E. N. MaNetHo, "Fault tolerance in distributed systems using rollback-recovery and process replication." Ph.D. dissertation, Rice University, Texas, Austin, October 1993

[11] ELNOZAHY E. N., ALVISI L., WANG Y. M., and JOHNSON D. B, "A survey of rollback-recovery protocols in message-passing systems." ACM Computing Surveys Volume 34, No 3, (September 2002): p. 375-408.

[12] ELNOZAHY E. N., JOHNSON D. B., and ZWAENEPOEL W., "The performance of consistent checkpointing." In Proc. of IEEE Symp. on Reliable Distributed Systems, (October 1992): p. 39-47.

[13] GERTH R. (August 1997) Concise Promela Reference. Eindhoven University. email: robg@win.tue.nl

[14] HELARY J. M., MOSTEFAOUI A., NETZER R. H., and RAYNAL M., Preventing useless checkpoints in distributed computations." In Proc. of the 16th Symposium on Reliable Distributed Systems, (1997): p. 183-190.

[15] HOLZMANN G. J. "The model checker SPIN." IEEE Trans.on Software Engineering, Volume 23 No 5, (May 1997): p. 279-295.

[16] HOLZMANN G. J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, September 2003.

[17] S. KALAISELVI and V. RAJARAMAN, "A survey of checkpointing algorithms for parallel and distributed computers." Sadhana, Volume 25, No. 5, (October 2000): p. 489-510.

[18] KOO R. and TOUEG S., "Checkpointing and rollback-recovery for distributed systems." IEEE Trans, on Software Engg. Volume 13, No 1, (1987): p. 23-31.

[19] LAI T. H. and YANG T. H., "On distributed snapshots." in Information Processing Letters Volume 25, (1987): p. 153-158.

[20] LAMPORT L. "Time, Clocks and the Ordering of Events in a Distributed System." Comm. ACM, Volume. 21, No. 7, (1978): pp. 558-565.

[21] LAMPORT L. and MELLER-SMITH. P. M., "Synchronizing clocks in the presence of faults." Journal of the ACM Volume 32, (January 1985.): p. 52-78.

[22] MANIVANNAN D. and SINGHAL M., "A low-overhead recovery technique using quasi-synchronous checkpointing." In Proc. of Distributed Computing Systems. ACM, (May 1996): p. 100-107.

[23] PRAKASH R. and SINGHAL M., "Low-cost checkpointing and failure recovery in mobile computing systems." IEEE Trans on Parallel and Distributed Systems Volume 7, No 10, (October 1996): p.1035-1048.

[24] RANDELL B., "System structure of software fault tolerance." IEEE Trans, on Software Engg. Volume 1, No 2, (1975): p. 220-23.

[25] RUYS T. C. SPIN Beginner's Tutorial. SPIN 2002 Workshop, University of Twente, Formal Methods and Tools Group. 11 April 2002.

[26] SCHLICHTING R.D. and SCHNEIDER F.B., "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems." ACM Trans Computer Systems, Volume 1, No.3, (1983): p. 222-238.

[27] SINGHAL M., SHIVARATRI N. G. Advanced Concepts in Operating Systems New Delhi: Tata McGraw-Hill 2005.

[28] SILVA L. M. and SILVA J. G., "The performance of coordinated and independent checkpointing." In Proc. of Intl. Parallel and Distributed Processing Symposium, (1999): p. 88-94.

[29] SISTLA A. P. and WELCH J. L., "Efficient distributed recovery using message logging." IEEE/ACM Trans on Networking Volume 4, No 5, (1996): p. 785-795.

[30] STROM R. E. and YEMINI S., "Optimistic recovery in distributed systems." ACM Trans. on Computer Systems Volume 3, No 3, (August 1985): p. 204-226.

[31] Spin Website: http://spinroot.com/spin/index.html

[32] TAMIR Y. and SEQUIN C. H., "Error recovery in multicomputers using global checkpoints." Intl. Conf. on Parallel Processing, (1984): p. 32-41.

[33] Tsai J., "Systematic comparisons of RDT communication-induced checkpointing protocols". In Proc. Of Pacific Rim International Symposium on Dependable Computing, 2004, pp: 66-75.

[34] VIEIRA G. M. D., GARCIA I.C., and BUZATO L. E. "Systematic analysis of index-based checkpointing algorithms using simulation." In Proc. of IX Brazilian Symp. On Fault-Tolerant Comput. (2001).

[35] WANG Y. M. LOWRY A. and FUCHS W. K., "Consistent global checkpoints that contain a given set of local checkpoints." IEEE Trans on Computers Volume 46, No 4, (April 1997): p. 456-468.

[36] WANG, Y. M., and FUCHS W. K., "Optimistic message logging for independent checkpointing in message passing systems." In Proc. of IEEE Symp. on Reliable Distributed Systems, (October 1992): p. 147-154.