A Review of Checkpointing Fault Tolerance Techniques in Distributed Mobile Systems

Rachit Garg[1], Praveen Kumar[2]

[1]Singhania University, Department of Computer Science & Engineering, Pacheri Bari (Rajasthan), India
[2]Meerut Institute of Engineering & Technology, Department of Computer Science & Engineering, Meerut (INDIA)-250005
{rachit.garg, pk223475}@yahoo.com

Abstract
Fault Tolerance Techniques enable systems to perform tasks in the presence of faults. A checkpoint is a local state of a process saved on stable storage. In a distributed system, since the processes in the system do not share memory, a global state of the system is defined as a set of local states, one from each process. In case of a fault in distributed systems, checkpointing enables the execution of a program to be resumed from a previous consistent global state rather than resuming the execution from the beginning. In this way, the amount of useful processing lost because of the fault is significantly reduced. Checkpointing is an effective fault tolerant technique in distributed system as it avoids the domino effect and require minimum storage requirement. Most of the earlier coordinated checkpoint algorithms block their computation during checkpointing and forces minimum-process or non-blocking even though many of them may not be necessary or non-blocking minimum-process but takes useless checkpoints or reduced useless checkpoint but has higher synchronization message overhead or has high checkpoint request propagation time. In this paper, we discuss various issues related to the checkpointing for distributed systems and mobile computing environments. We also present a survey of some checkpointing algorithms for distributed systems.

KEYWORDS
Fault tolerance, coordinated checkpointing, consistent global state, and mobile distributed system.

1. Introduction

Parallel computing with clusters of workstations (cluster computing) is being used extensively as they are cost-effective and scalable, and are able to meet the demands of high performance computing. Increase in the number of components in such systems increases the failure probability. To provide fault tolerance it is essential to understand the nature of the faults that occur in these systems. There are mainly two kinds of faults: permanent and transient. Permanent faults are caused by permanent damage to one or more components and transient faults are caused by changes in environmental conditions. Permanent faults can be rectified by repair or replacement of components. Transient faults remain for a short duration of time and are difficult to detect and deal with. Hence it is necessary to provide fault tolerance particularly for transient failures in parallel computers. Fault-tolerant techniques enable a system to perform tasks in the presence of faults. Fault tolerance involves fault detection, fault location, fault containment and fault recovery. Fault tolerance can be provided in a parallel computer at three different levels: hardware level, architecture level and application/system software level. In the hardware and architecture levels, importance is given to fault detection and replication of tasks. In the application/system software level, checkpointing techniques are used to provide fault tolerance. It is easier and more cost effective to provide software fault tolerance solutions than hardware solutions to cope with transient failures. Thus, checkpointing is an important technique to ensure software fault tolerance.

Fault Tolerance Techniques enable systems to perform tasks in the presence of faults. The likelihood of faults grows as systems are becoming more complex and applications are requiring more resources, including execution speed, storage capacity and communication bandwidth. Reliability and resilience are critical issues in parallel and distributed systems. These systems comprise of various computing devices and communication and storage resources. There are a number of fault sources in a system, including physical failure of components, environmental interference, software errors, security violations, and operator errors. Faults can be classified into two types: permanent and transient faults. Permanent faults are faults that cause a permanent damage to some part of the system. Recovery from permanent faults must include replacement of the damaged part and reconfiguration of the system. Transient faults are short-lived and do not lead to permanent damage. Recovery from transient

faults is comparatively simple as compared to the permanent faults, because reconfiguration of the system is not needed. Generally, the detection of the transient faults is more difficult, because they may disappear without a detectable effect of the system.
In scientific and commercial applications, in case of a detection of a transient fault, the execution of the program needs to be interrupted and resumed from beginning. As a result, the big applications are completed only if a sufficiently long fault-free interval of time exists in the system. In the presence of faults, the average execution of the program may grow exponentially with the length of the program. Checkpointing is primarily used to avoid losing all the useful processing done before a fault has occurred. Checkpointing consists of intermittently saving the state of a program in a reliable storage medium. Upon detection of a fault, previous consistent state is restored. In case of a fault, checkpointing enables the execution of a program to be resumed from a previous consistent state rather than resuming the execution from the beginning. In this way, the amount of useful processing lost because of the fault is significantly reduced. With checkpointing, the average execution of a program grows only linearly with the length of the program [8].

2. System Model

A distributed system consists of number of processes P_1 , P_2 , P_3 , ... P_n , which communicate only through messages. Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively. Figure 1 below shows a system consisting of three processes and interactions with the outside world.

Rollback recovery protocols generally make assumptions about the reliability of the inter-process communication. Some protocols assume that the communication subsystem delivers messages reliably, in first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can lose, duplicate or reorder messages. The choice between these two assumptions usually affects the complexity of checkpointing and failure recovery. A generic correctness condition for rollback recovery can be defined as follows: “a system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure”. Rollback recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.

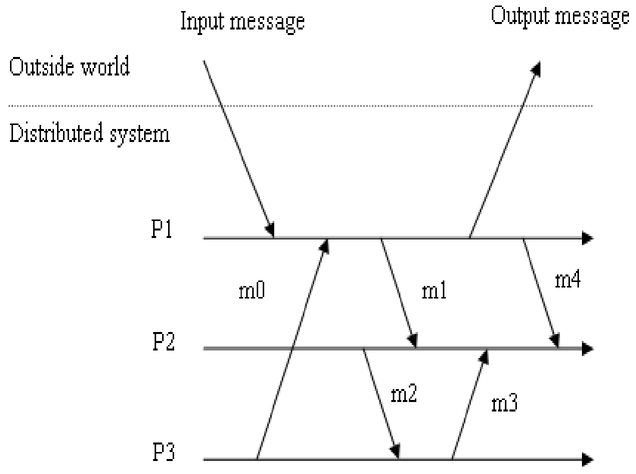

Figure 1 Distributed System with three processes

In distributed systems all processes save their local states at certain instants of time. This saved state is known as a local checkpoint. A 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. Check-pointing is the process of saving the status information. By periodically invoking the check-pointing process, one can save the status of a program at regular intervals. If there is a failure one may restart computation from the last check point thereby avoiding repeating computations from the beginning. The process of resuming computation by rolling back to a saved state is called rollback recovery. Events in uni-processor are governed by a single clock, providing total ordering of events. When an error is detected all the events after the last check point are repeated. Check-pointing becomes a real concern in case of Distributed Systems (or Parallel Systems or Multiprocessor Systems) because there are multiple streams of execution and there is no global clock. The absence of global clock makes it difficult to initiate check points in all the streams of execution at the same time instance. We have to pick one checkpoint from each stream in such a way that the set of these check points are concurrent.

Local checkpoint is the saved state of a process at a processor at a given instance. Global checkpoint is a collection of local checkpoints, one from each process. A global state is said to be ‘consistent’ if it contains no orphan message (A Message whose receive event is recorded, but its send event is lost).

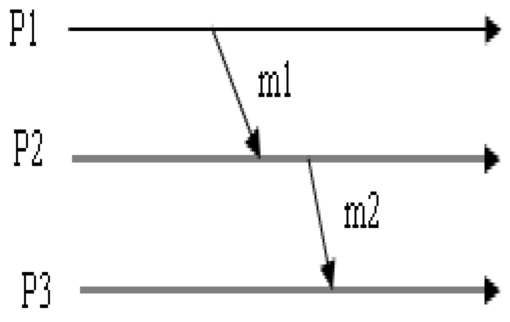

Figure 2 Depending of Processes

In Figure 2, P_3 receives message m_2 sent by process P_2. We say that P_3 is directly dependent upon P_2. Similarly P_2 is directly dependent on P_1 due to m1. We can also say that P_3 is transitively dependent upon P_1. In this case, if P_3 takes its checkpoint after processing m_2, then P_2 should take its checkpoint after sending m_2, otherwise m_2 will become orphan. Similarly P_1 should take its checkpoint after sending m_1 otherwise m_1 will become orphan.

An example of failure recovery

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure recovery.

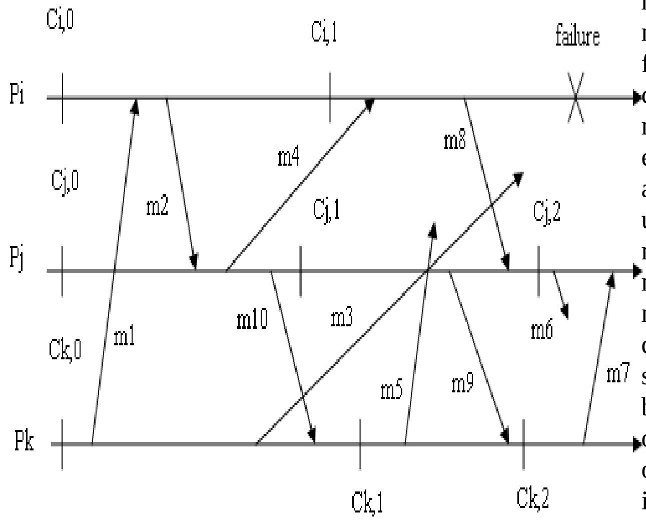

Figure 3 Failure Recovery

We now describe the issues involved in a failure recovery with the help of a distributed computation shown in figure 3. The computation comprises of three processes P_i, P_j and P_k connected through a communication network. The processes communicate solely by exchanging messages over fault-free, FIFO communication channels. Processes P_i, P_j and P_k have taken checkpoints { $C_{i,0}$, $C_{i,1}$ }, { $C_{j,0}$, $C_{j,1}$, $C_{j,2}$ } and { $C_{k,0}$, $C_{k,1}$ } respectively and these processes have exchanged messages m_1 to m_{10} as shown in figure 1.4. Suppose process Pi fails at the instance indicated in the figure. All the contents of the volatile memory of Pi are lost and after Pi has recovered from the failure, the system needs to be restored to a consistent global state from where the processes can resume their execution. Process P_i's state is restored to a valid state by rolling it back to its most recent checkpoint $C_{i,1}$. To restore the system to a consistent state, the process P_j rolls back to checkpoint $C_{j,1}$ because the rollback process P_i to checkpoint $C_{i,1}$, created an orphan message m_8 (the receive event of m_8 is recorded at process P_j while the send event of m_8 has been undone at process P_i). Note that process P_j does not roll back to checkpoint $C_{j,2}$ but to checkpoint $C_{j,1}$, because rolling back to checkpoint $C_{j,2}$ does not eliminate the orphan message m_8 . Even this resulting state is not a consistent global state as an orphan message m9 is created due to the roll back of process P_j to checkpoint $C_{j,1}$. To eliminate this orphan message, process P_k rolls back to checkpoint $C_{k,1}$. The restored global state {$C_{i,1}$, $C_{j,1}$, $C_{k,1}$ } is a consistent state as it is free from orphan messages. Although the system has been restored to a consistent state, several messages are left in an erroneous state which must be handled correctly.

Messages m_1 , m_2 , m_4 , m_7 , m_8 , m_9 , m_{10} had been received at the points indicated in the figure and messages m_3 , m_5 and m_6 were in transit when the failure occurred. Restoration of system state to checkpoints {$C_{i,1}$, $C_{j,1}$, $C_{k,1}$ } automatically handles messages m_1 , m_2 , m_{10} because the send and receive events of messages m_1 , m_2 , m_{10} have been recorded and both the events m_7 , m_8 , m_9 have been completely undone. These messages cause no problem and we call messages m_1 , m_2 , m_{10} normal messages and messages m_7 , m_8 , m_9 vanished messages. Messages m_3 , m_4 , m_5 , m_6 are potentially problematic. Message m_3 is in transit during the failure and it is a delayed message m_3 has several possibilities: m_3 might arrive at process P_i before it recovers, it might arrive while P_i is recovering or it might arrive after P_i has completed recovery. Each of these cases must be dealt with correctly. Message m_4 is a lost message since the send for m_4 is recorded in the restored state for process P_j , but the receive event has been undone at process P_i . Process P_j will not resend m_4 without an additional mechanism, since the send m_4 at P_j occurred before the checkpoint and the communication system successfully delivered m_4 . Messages m_5 and m_6 are delayed orphan messages and pose perhaps the most serious problem of all the messages. When messages m_5 and m_6 arrive at their respective destinations, they must be discarded since their send events have been undone. Processes after resuming execution from their checkpoints, will generate both of the messages, and recovery techniques must be able to distinguish between messages like m_3 and those like m_5 and m_6 .

Lost messages like m_4 can be handled by having processes keep a message log of all the sent messages. So when a process restores to a checkpoint, it replays the messages from its log to handle the lost message problem. However, message logging and message replaying during recovery can result in duplicate messages. In the example shown in figure 3 when process P_j replays messages from its log, it will regenerate message m_{10} . Process P_k which has already received message m_{10} , will receive it again, thereby causing inconsistency in the system state. Therefore, these duplicate messages must be handled properly.

Overlapping failures further complicate the recovery process. A process P_j that begins rollback/recovery in response to the failure of a process P_i can itself fail and develop amnesia with respect process P_i 's failure; that is process P_j can act in a fashion that exhibits ignorance of process P_i 's failure. If overlapping failures are to be tolerated, a mechanism must be introduced to deal with amnesia and the resulting inconsistencies.

3. Checkpointing-based Rollback Recovery

A checkpoint is a local state of a process saved on stable storage. In a distributed system, since the processes in the system do not share memory, a global state of the system is defined as a set of local states, one from each process. The state of channels corresponding to a global state is the set of messages sent but not yet received. A lost or in-transit message is one, the sending of which has been recorded by the sender but whose receiving could not be recorded by the receiving process. An orphan message is a message whose receive event is recorded, but its send event is lost. A global state is
said to be "consistent" if it contains no orphan message and all the in-transit messages are logged.

After a failure, a system must be restored to a consistent system state. Essentially, a system state is consistent if it could have occurred during the preceding execution of the system from its initial state, regardless of the relative speeds of individual processes. This assumes that the total execution of the system is equivalent to some fault free execution [8]. It has been shown that two local checkpoints being causally unrelated is a necessary but not sufficient condition for them to belong to the same consistent global checkpoint. This problem was first addressed by Netzer and Xu who introduced the notion of a Z-path between local checkpoints to capture both their causal and hidden dependencies [37]. Considering a checkpoint and communication pattern, the rollback dependency track ability property stipulates that there is no hidden dependency between local checkpoints [11]. To be able to recover a system state, all of its individual process states must be able to be restored. A consistent system state in which each process state can be restored is thus called a recoverable system state.

Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and messages it sends and receives; it can record nothing else. To determine a global system state, a process Pi must enlist the cooperation of other processes that must record their own local states and send the recorded local states to Pi. All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global state detection algorithm is to be superimposed on the underlying computation; it must run concurrently with, but not alter, this underlying computation [19].

The state detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds- a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. All snapshots cannot be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed. Yet, the composite picture should be meaningful. The problem before us is to define meaningful and then to determine how the photographs should be taken [19].

The problem of taking a checkpoint in a message passing distributed system is quite complex because any arbitrary set of checkpoints cannot be used for recovery [19], [50], [51]. This is due to the fact that the set of checkpoints used for recovery must form a consistent global state.

In backward error recovery, depending on the programmer's intervention in process of checkpointing, the classification can be:

User triggered checkpointing schemes require user interaction and are useful in reducing the stable storage requirement [22]. These are generally employed where the user has the knowledge of the computation being performed and can decide the location of the checkpoints. The main problem is the identification of the checkpoint location by a user. The transparent checkpointing techniques do not require user interaction and can be classified into following categories:

In uncoordinated or independent checkpointing, processes do not coordinate their checkpointing activity and each process records its local checkpoint independently [14], [56], [64]. It allows each process the maximum autonomy in deciding when to take checkpoint, i.e., each process may take a checkpoint when it is most convenient. It eliminates coordination overhead all together and forms a consistent global state on recovery after a fault [14]. After a failure, a consistent global checkpoint is established by tracking the dependencies. It may require cascaded rollbacks that may lead to the initial state due to domino-effect [31], [50], [51]. It requires multiple checkpoints to be saved for each process and periodically invokes garbage collection algorithm to reclaim the checkpoints that are no longer needed. In this scheme, a process may take a useless checkpoint that will never be a part of global consistent state. Useless checkpoints incur overhead without advancing the recovery line [22]. The main disadvantage of this approach is the domino-effect. For example, in figure 4, the latest set of checkpoints {C_{12} , C_{22}, C_{32} } is not consistent.

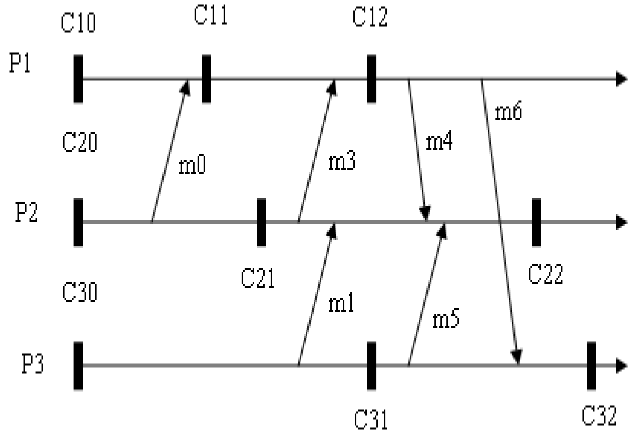

Figure 4 Uncoordinated Checkpointing

Because, the constituted global state contains orphan messages m4 and m6. However, the global state {C_{11}, C_{21}, C_{31}} is consistent. It allows each process to have maximum autonomy in deciding when to take checkpoint. Each process may take a checkpoint when it is most convenient. A process may reduce the overhead by taking checkpoints when the amount of state information to be saved is small. After a failure, a consistent global checkpoint is established by tracking the dependencies. It may require cascaded rollbacks that may lead to the initial state due to domino effect. A checkpoint that cannot belong to any consistent global check point is called useless checkpoint. One way to avoid domino effect is to clean the system from these useless checkpoints. It requires multiple checkpoints to be saved for each process and periodically invokes garbage collection algorithm to reclaim the checkpoints that are no longer needed. In this scheme, a process may take a useless check point that will never be a part of global consistent state. Also useless checkpoints incur overhead without advancing the recovery line.

In coordinated or synchronous checkpointing, processes take checkpoints in such a manner that the resulting global state is consistent. Mostly it follows two-phase commit structure [19], [23], [31]. In the first phase, processes take tentative checkpoints and in the second phase, these are made permanent. The main advantage is that only one permanent checkpoint and at most one tentative checkpoint is required to be stored. In case of a fault, processes rollback to last checkpointed state. A permanent checkpoint can not be undone. It guarantees that the computation needed to reach the checkpointed state will not be repeated. A tentative checkpoint, however, can be undone or changed to be a permanent checkpoint.

A straightforward approach to coordinated checkpointing is to block communications while the checkpointing protocol executes [58]. A coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives the message, it stops its executions, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgement message back to the coordinator. After the coordinator receives acknowledgements from all processes, it broadcasts a commit message that completes the two-phase checkpoint protocol. On receiving commit, a process converts its tentative checkpoint into permanent one and discards its old permanent checkpoint, if any. The process is then free to resume execution and exchange messages with other processes.

The coordinated checkpointing protocols can be classified into two types: blocking and non-blocking. In blocking algorithms, as mentioned above, some blocking of processes takes place during checkpointing [31], [58]. In non-blocking algorithms, no blocking of processes is required for checkpointing [19], [23]. The coordinated checkpointing algorithms can also be classified into following two categories: minimum-process and all process algorithms. In all-process coordinated checkpointing algorithms, every process is required to take its checkpoint in an initiation [19], [23]. In minimum-process algorithms, minimum interacting processes are required to take their checkpoints in an initiation [31].

Communication-induced checkpointing avoids the domino-effect without requiring all checkpoints to be coordinated [12], [26], [35]. In these protocols, processes take two kinds of checkpoints, local and forced. Local checkpoints can be taken independently, while forced checkpoints are taken to guarantee the eventual progress of the recovery line and to minimize useless checkpoints. As opposed to coordinated checkpointing, these protocols do no exchange any special coordination messages to determine when forced checkpoints should be taken. But, they piggyback protocol specific information [generally checkpoint sequence numbers] on each application message; the receiver then uses this information to decide if it should take a forced checkpoint. This decision is based on the receiver determining if past communication and checkpoint patterns can lead to the creation of useless checkpoints; a forced checkpoint is taken to break these patterns [22], [35].

4. Log-Based Rollback Recovery

Message-logging protocols (for example [3], [4], [5], [6], [9], [24], [25], [27], [47], [57], [59], [60], [61], [62], are popular for building systems that can tolerate process crash failures. Message logging and checkpointing can be used to provide fault tolerance in distributed systems in which all inter-process communication is through messages. Each message received by a process is saved in message log on stable storage. No coordination is required between the checkpointing of different processes or between message logging and checkpointing. The execution of each process is assumed to be deterministic between received messages, and all processes are assumed to execute on fail stop processes.

When a process crashes, a new process is created in its place. The new process is given the appropriate recorded local state, and then the logged messages are replayed in the order the process originally received them. All message-logging protocols require that once a crashed process recovers, its state needs to be consistent with the states of the other processes [22], [65]. This consistency requirement is usually expressed in terms of orphan processes, which are surviving processes whose states are inconsistent with the recovered states of crashed processes. Thus, message- logging protocols guarantee that upon recovery, no process is an orphan. This requirement can be enforced either by avoiding the creation of orphans during an execution, as pessimistic protocols do, or by taking appropriate actions during recovery to eliminate all orphans as optimistic protocols do. Bin Yao et al. [65] describes a receiver based message logging protocol for mobile hosts, mobile support stations and home agents in a Mobile IP environment, which guarantees independent recovery. Checkpointing is utilized to limit log size and recovery latency.

Pessimistic logging protocols are designed under the assumption that a failure can occur after any nondeterministic event in the computation. This assumption is “pessimistic” since in reality failures are rare. In their most straightforward form, pessimistic protocols log to stable storage the determinant of each nondeterministic event before the event is allowed to affect the computation. These pessimistic protocols implement the following property, often referred to as *synchronous logging*, which is a strengthening of the always-no-orphans condition: This property stipulates that if an event has not been logged on stable storage, then no process can depend on it. In addition to logging determinants, processes also take periodic checkpoints to limit the amount of work that has to be repeated in execution replay during recovery. Should a failure occur when the application program is restarted from the most recent checkpoint and the logged determinants are used during recovery to recreate the pre-failure execution. This property has four advantages: i) Processes can commit output to the outside world without running a special protocol. ii) Processes restart from their most recent checkpoint upon a failure, therefore limiting the extent of execution that has to be replayed. Thus, the frequency of checkpoints can be determined by trading off the desired runtime performance with the desired protection of the on-going execution. iii) Recovery is simplified because the effects of a failure are confined only to the processes that fail. Functioning processes continue to operate and never become orphans because a process always recovers to the state that included its most recent interaction with any other process or with the outside world. This is highly desirable in practical systems. iv) Recovery information can be garbage-collected easily. Older checkpoints and determinants of nondeterministic events that occurred before the most recent checkpoint can be reclaimed because they will never be needed for recovery [22].

Optimistic logging protocols processes log determinants *asynchronously* to stable storage. These protocols make the optimistic assumption that logging will complete before a failure occurs. Determinants are kept in a volatile log, which is periodically flushed to stable storage. Thus, optimistic logging does not require the application to block waiting for the determinants to be actually written to stable storage, and therefore incurs little overhead during failure-free execution. However, this advantage comes at the expense of more complicated recovery, garbage collection, and slower output commit than in pessimistic logging. If a process fails, the determinants in its volatile log will be lost, and the state intervals that were started by the nondeterministic events corresponding to these determinants cannot be recovered. Furthermore, if the failed process sent a message during any of the state intervals that cannot be recovered, the receiver of the message becomes an orphan process and must roll back to undo the effects of receiving the message. Optimistic protocols [22] do not implement the *always-no-orphans* condition, and therefore permit the temporary creation of orphan processes. To perform these rollbacks correctly, optimistic logging protocols track causal dependencies during failure-free execution. Upon a failure, the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan.

Causal logging [22] has the failure-free performance advantages of optimistic logging while retaining most of the advantages of pessimistic logging. Like optimistic logging, it avoids synchronous access to stable storage except during output commit. Like pessimistic logging, it allows each process to commit output independently and never creates orphans, thereby isolating processes from the effects of failures that occur in other processes. Furthermore, causal logging limits the rollback of any failed process to the most recent checkpoint on stable storage. This reduces the storage overhead and the amount of work at risk.

5. Earlier Work

Gupta, B., and Rahimi, S.[68], have addressed the complex problem of recovery for concurrent failures in distributed computing environment. They have proposed a new approach in which they have effectively dealt with both orphan and lost messages. The proposed checkpointing and recovery approaches enable each process to restart from its recent checkpoint and hence guarantee the least amount of recomputation after recovery. It also means that a process needs to save only its recent local checkpoint. In this regard, they have introduced two new ideas. First, the proposed value of the common checkpointing interval was such that it enables an initiator process to log the minimum number of messages sent by each application process. Second, the determination of the lost messages was always done a priori by an initiator process; besides this was done while the normal distributed application was running. This was quite meaningful because it does not delay the recovery approach in any way.

They have proposed a checkpointing approach that was a single phase one and non-blocking in nature; besides it does not have any synchronization delay. It makes sure that at the time of recovery they do not have to deal with orphan messages unlike many of the existing works and also processes can restart from their respective recent checkpoints. The choice of the value of the common checkpointing interval enables to use as little information related to the lost and delayed messages as possible for consistent operation after the system restarts. The determination of the lost messages was always done a priori by an initiator process; besides it was done while the normal distributed application was running. It was meaningful because it does not delay the recovery approach in any way. Besides, the recovery approach was independent of the number of processes that may fail concurrently. Finally note that their checkpointing and recovery schemes are independent of the effect of any clock drift on the respective sequence numbers of the recent checkpoints of the processes, because they consider only processes' recent checkpoints irrespective of their sequence numbers.

Biswas, S., and Neogy, S.[69], proposed a new checkpointing and failure recovery algorithm for mobile computing system. Mobile hosts save checkpoints based on mobility and movement patterns. Movement patterns considered here are of three types – i) Intercell movement pattern ii) combination movement pattern ii) Intracell movement pattern. Mobile hosts save checkpoints when number of hand-off exceeds a predefined hand-off threshold value. Disconnection was a frequent phenomenon and was of two types: i) planned disconnection ii) unplanned disconnection. Hence mobile hosts save two types of checkpoints - i) permanent checkpoint based on hand-off threshold value covering unplanned disconnection ii) migration checkpoint covering planned disconnection. Hand-off threshold was a function mobility rate, movement pattern, message passing frequency and failure rate.

They proposed checkpointing algorithm which was in comparison with other relevant works because it was designed based not only on mobility and hand-off of MHs but movement patterns were also considered. Unike others, MHs moving within a cell was checkpointed exclusively. Hence, their checkpointing scheme was stronger from the point of view of failure recovery. Disconnection of MHs was a frequent phenomenon which may delay checkpointing. Hence the concept of migration checkpoint was introduced before planned disconnection so that checkpointing can be completed without any dealy resulting enhanced fault tolerance in the proposed scheme.

Prakash-Singhal [45] have stated that a checkpointing algorithm for mobile distributed systems should have following characteristics: (i) It should be minimum process (ii) it should be non-intrusive (iii) it should not awake the MHs in "doze mode operations". They proposed a synchronous snapshot collection algorithm for mobile systems that neither forces every node to take a local snapshot, nor blocks the underlying computation during snapshot collection. If a node initiates snapshot collection, local snapshots of only those nodes that have directly or transitively affected the initiator since their last snapshots need to be taken. The global snapshot collection terminates within a finite time of its invocation. They also proposed a minimal rollback recovery algorithm in which the computation at a node is rolled back only if it depends on operations that have been undone due to the failure of node(s). Both the algorithms have low communication and storage overheads and meet the low energy consumption and low bandwidth constraints of mobile computing systems. An interesting aspect of their algorithm is that it has a lazy phase that enables nodes to take local snapshots in a quasi-asynchronous fashion, after the coordinate snapshot collection phase the aggressive phaser is over. This further reduces the amount of computation that is rolled back during recovery from node failures. Moreover the lazy phase advances the checkpoint slowly rather than in a burst. This avoids contention for the low bandwidth channels. Basically they have proposed a minimum process non-blocking checkpointing algorithm. Cao-Singhal [16] has shown that the algorithm [45] may lead to inconsistencies.

Cao and Singhal [16] presented a minimum process checkpointing algorithm in which the dependency information is recorded by a Boolean vector. This algorithm is a two phase protocol and saves two kinds of checkpoints on the stable storage. In the first phase the initiator sends a request to all processes to send their dependency vectors. On receiving the request each process sends its dependency vector. Having received all the dependency vectors, the initiator constructs an N x N dependency matrix with one row per process, represented by the dependency vector of the process. Based on the dependency matrix, the initiator can locally calculate all the processes on which the initiator transitively depends. After the initiator finds all the process that need to take checkpoints and adds them to the set S forced and ask them to take checkpoints. Any process receiving a checkpoint request takes the checkpoint and sends a reply. The process has to be blocked after receiving the dependency vectors request and resumes its computation after receiving a checkpoint request.

Cao-Singhal [17] proposed a minimum process coordinated checkpointing algorithm for mobile distributed systems. They introduced the concept of "mutable checkpoint", which is neither a tentative checkpoint nor a permanent checkpoint. It is saved on MH. The basic idea of the algorithm is as follows. In the first phase the initiator process says P sends the checkpoint request to P1 to P10 is directly dependent upon P. On getting the checkpointing request, P1 takes the following actions: (i) P1 takes its tentative checkpoint (ii) it finds the processes which are in its dependency vector but not in the minimum set received from the P10 – P1 sends the checkpointing request to such processes. Suppose P1 sends m to P1 after taking its tentative checkpoint. When P1 receives m and finds that it has not taken its tentative checkpoint for the current initiation, it cannot conclude whether it will be included in the minimum set in the current initiation. In this case, if P1 takes its tentative checkpoint after receiving m, m will become orphan. Therefore P1 takes its mutable checkpoint before processing m, if P1 gets the checkpointing request, it converts its mutable checkpoint into tentative checkpoint; otherwise, at the time of commit. P1 discards its mutable checkpoint.

Weigang et al [64] presented a coordinated non-blocking algorithm for distributed mobile systems. They proposed to reduce the MHs coordination message overhead by introducing an idea called proxy coordinator. The proxy coordinator is a process which is running on the MSS. When a process initiate the checkpointing operation, it takes its tentative checkpoint and sends the checkpointing request to all the dependent processes through its MSS. On receiving the checkpointing request by the initiator MSS a process called proxy coordinator is started on this MSS. This proxy coordinator further coordinates the checkpointing process on behalf of the initiator process. They assumed that a process will not receive a checkpoint request associated with another initiator before the current executing one is completed. They shown that Cao-Singhal algorithm [16] may lead to inconsistencies during concurrent initiations.

Kumar-et al [32] proposed a non-blocking checkpointing algorithm based on keeping track of direct dependencies of processes. Each process maintains a direct dependency vector. In their scheme, initiator process collects the direct dependency vectors of all processes, computes minimum set, and sends the checkpoint request along with the minimum set to relevant processes. This reduces the time to take the checkpoints. If new dependencies are created during checkpointing process, those are updated and updated minimum set is formed.

Wang and Fuchs [65] proposed a coordinated checkpointing scheme in which they incorporated the technique of lazy checkpoint coordination into an uncoordinated checkpointing protocol for bounding rollback propagation Recovery line progression is made by performing communication induced checkpoint coordination only when predetermined consistency criterion is violated. The notation of laziness provides a trade off between extra checkponts during normal execution and average rollback distance for recovery.

L K Awasthi-Kumar [33] proposed a minimum process coordinated checkpointing protocol for mobile distributed systems. Where the number of useless checkpoint and the blocking of processes are reduced using the probabilistic approach and by computing the tentative minimum set in the beginning. This algorithm is the first one to combine and non-blocking scheme in one algorithm.

Gupta et al. [54] proposed a single phase non-blocking coordinated checkpointing approach for mobile computing environment. In their algorithm, the processes are allowed to take the permanent checkpoints directly without taking tentative checkpoints and whenever a process is busy, the process takes a checkpoint after the completion of current procedure. However, this scheme has the disadvantage that it does consider the case of failure during the checkpointing operation which may result in the inconsistent states of the processes.

Mannivannan and Singhal proposed a quasi synchronous checkpointing algorithm [35]. This algorithm is simple and has a merit of asynchronous checkpoint low overhead, and a merit of synchronous checkpointing low recovery time. In this algorithm, each process takes checkpoint independently called basic checkpoints. Checkpoints triggered by message reception are called forced checkpoints. The checkpoint index is increased by one after taking a basic or forced checkpoint. When process P1 receives a message m, with piggybacked information index, from process P1 and P1s index, is small than Index, a forced checkpoint is taken to advance the recovery line. Although the algorithm has a low checkpoint overhead, it has to maintain multiple checkpoints.

In minimum-process coordinated checkpointing, some processes may not checkpoint for several checkpoint initiations. In the case of a recovery after a fault, such processes may rollback to far earlier checkpointed state and thus may cause greater loss of computation. In all-process coordinated checkpointing, the recovery line is advanced for all processes but the checkpointing overhead may be exceedingly high. To optimize both matrices, the checkpointing overhead and the loss of computation on recovery, P.Kumar [66] proposed a hybrid checkpointing algorithm, wherein an all-process coordinated checkpoint is taken after the execution of minimum-process coordinated checkpointing algorithm for a fixed number of times. Thus, the Mobile nodes with low activity or in doze mode operation may not be disturbed in the case of minimum-process checkpointing and the recovery line is advanced for each process after an all-process checkpoint. Additionally, he tried to minimize the information piggybacked onto each computation message. For minimum-process checkpointing, he designed a blocking algorithm, where no useless checkpoints are taken and an effort has been made to optimize the blocking of processes. He proposed to delay selective messages at the receiver end. By doing so, processes are allowed to perform their normal computation, send messages and partially receive them during their blocking period. The proposed minimum-process blocking algorithm forces zero useless checkpoints at the cost of very small blocking.

Kumar and Kumar [67] proposed an algorithm which is based on keeping track of direct dependencies of processes. Initiator MSS collects the direct dependency vectors of all processes, computes the tentative minimum set (minimum set or its subset), and sends the checkpoint request along with the tentative minimum set to all MSSs. This step is taken to reduce the time to collect the coordinated checkpoint. It will also reduce the number of useless checkpoints and the blocking of the processes. Suppose, during the execution of the checkpointing algorithm, *Pi* takes its checkpoint and sends *m* to *Pj*. *Pj* receives *m* such that it has not taken its checkpoint for the current initiation and it does not know whether it will get the checkpoint request. If *Pj* takes its checkpoint after processing *m*, *m* will become orphan. In order to avoid such orphan messages, they propose the following technique. If *Pj* has sent at least one message to a process, say *Pk* and *Pk* is in the tentative minimum set, there is a good probability that *Pj* will get the checkpoint request. Therefore, *Pj* takes its induced checkpoint before processing *m*. An induced checkpoint is similar to the mutable checkpoint [15]. In this case, most probably, *Pj* will get the checkpoint request and its induced checkpoint will be converted into permanent one. There is a less probability that *Pj* will not get the checkpoint request and its induced checkpoint will be discarded. Alternatively, if there is not a good probability that *Pj* will get the checkpoint request, *Pj* buffers *m* till it takes its checkpoint or receives the commit message. They have tried to minimise the number of useless checkpoints and blocking of the process by using the probabilistic approach and buffering selective messages at the receiver end. Exact dependencies among processes are maintained. It abolishes the useless checkpoint requests and reduces the number of duplicate checkpoint requests.

Neogy, S.[70], presented a proposal for achieving fault tolerance in wireless and mobile computing systems and proposed that it saves the nodes possible retransmission thereby lowering network traffic. This approach particularly tackles the situation of intermittent failures due to disconnection, wireless channel saturation with traffic, low power of devices in wireless/mobile computing environment, though coordinated checkpointing was employed here but it does not force all local computation units to take checkpoints at every initiation thereby saving power and communication overhead in wireless network. She found that it was possible to have such an architecture even in a wireless system without any extra overhead added.

Pourmahmoud, S., Asbaghi, S., and Haghighat., A.T.[71], discussed on the size of rollback it has in the presence of failures. In order to determining the recovery line in checkpoint-based recovery, they first studied common approaches: dependency graph and checkpoint graph and provide some algorithms for these approaches. Then they introduced a new approach for calculating the recovery line and making a graph (independent graph). Finally they presented a solution for reducing the cost of graph when calculating the recovery line, particularly when the domino effect is occurred. They reviewed some approaches for calculating the recovery line in uncoordinated checkpointing. They introduced a new approach for reducing this cost of graph when calculating the recovery line (independent graph). Also another method was also presented and this was more useful when there was the domino effect. First, it recognized the useless checkpoints and dose not take them in graph building so the resulting graph will be smaller. They introduced this method and tried to reduce the overhead of distributed systems in recovery line detection when a failure occurs.

6. CONCLUSION

Various Fault tolerance solutions can be implemented in a variety of forms. They include software libraries, special programming languages, compiler or preprocessor modifications, operating system extensions, and system middleware. Each method has its own tradeoffs in terms of power, portability, and ease of use. A survey literature on checkpointing algorithm shows that a large number of papers have been published. A majority of these algorithms are based on the article by Chandy & Lamport (1985) and have Checkpointing algorithms for parallel and distributed computing been obtained by relaxing many of the assumptions made by them. The main aim of improving the earlier extensions of the Chandy & Lamport (1985) algorithms was to minimize the overhead of coordination between processes in a multiprocessor system. More recent published work attempts to minimise the context-saving overhead.

References

[1] Acharya A. and Badrinath B. R., “Checkpointing Distributed Applications on Mobile Computers,” *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pp. 73-80, September 1994.

[2] Acharya A., “Structuring Distributed Algorithms and Services for networks with Mobile Hosts”, *Ph.D. Thesis, Rutgers University*, 1995.

[3] Alvisi, Lorenzo and Marzullo, Keith,“ Message Logging: Pessimistic, Optimistic, Causal, and Optimal”, *IEEE Transactions on Software Engineering*, Vol. 24, No. 2, February 1998, pp. 149-159.

[4] L. Alvisi, Hoppe, B., Marzullo, K., “Nonblocking and Orphan-Free message Logging Protocol,” *Proc. of 23rd Fault Tolerant Computing Symp.*, pp. 145-154, June 1993.

[5] L. Alvisi,“ Understanding the Message Logging Paradigm for Masking Process Crashes,“ *Ph.D. Thesis, Cornell Univ., Dept. of Computer Science*, Jan. 1996. Available as Technical Report TR-96-1577.

[6] L. Alvisi and K. Marzullo,“ Tradeoffs in implementing Optimal Message Logging Protocol”, *Proc. 15th Symp. Principles of Distributed Computing*, pp. 58-67, ACM, June, 1996.

[7] Adnan Agbaria, Wiilliam H Sanders,“ Distributed Snapshots for Mobile Computing Systems”, *IEEE Intl. Conf. PERCOM’04*, pp. 1-10, 2004.

[8] Avi Ziv and Jehoshua Bruck, “ Checkpointing in Parallel and Distributed Systems”, *Book Chapter from Parallel and Distributed Computing Handbook edited by Albert Z. H. Zomaya*, pp. 274-302, Mc Graw Hill, 1996.

[9] A. Borg, J. Baumbach, and S. Glazer,“ A Message System Supporting Fault Tolerance”, *Proc. Symp. Operating System Principles*, pp. 90-99, ACM SIG OPS, Oct. 1983.

[10] Adnan Agbaria, William H. Sanders, “ Distributed Snapshots for Mobile Computing Systems”, *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications (Percom’04)*, pp. 1-10, 2004.

[11] Baldoni R., Hélary J-M., Mostefaoui A. and Raynal M., “ Rollback Dependency Trackability: A Minimial Characterization and its Protocol”, *Information and Computation*, 165, pp. 144-173, 2003.

[12] Baldoni R., Hélary J-M., Mostefaoui A. and Raynal M., “A Communication- Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability,” *Proceedings of the International Symposium on Fault-Tolerant-Computing Systems*, pp. 68-77, June 1997.

[13] Bhagwat P., and Perkins, C.E., “A mobile Networking System based on Internet Protocol (IP)”,*USENIX Symposium on Mobile and Location-Independent Computing,* August 1993.

[14] Bhargava B. and Lian S. R., “Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems-An Optimistic Approach,” *Proceedings of 17th IEEE Symposium on Reliable Distributed Systems*, pp. 3-12, 1988.

[15] Cao G. and Singhal M., “On coordinated checkpointing in Distributed Systems”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no.12, pp. 1213-1225, Dec 1998.

[16] Cao G. and Singhal M., "On the Impossibility of Min-process Non-blocking Checkpointing and an Efficient Checkpointing Algorithm for Mobile Computing Systems," *Proceedings of International Conference on Parallel Processing*, pp. 37-44, August 1998.

[17] Cao G. and Singhal M., "Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing systems," *IEEE Transaction On Parallel and Distributed Systems*, vol. 12, no. 2, pp. 157-172, February 2001.

[18] Cao G. and Singhal M., "Checkpointing with Mutable Checkpoints", *Theoretical Computer Science*, 290(2003), pp. 1127-1148.

[19] Chandy K. M. and Lamport L., "Distributed Snapshots: Determining Global State of Distributed Systems," *ACM Transaction on Computing Systems*, vol. 3, No. 1, pp. 63-75, February 1985.

[20] F. Cristian and F. Jahanian, " A timestamp-based Checkpointing Protocol for Long Lived Distributed Computations", *Proc IEEE Symp. Reliable Distributed Systems*, pp. 12-20, 1991.

[21] Dieter Kranzlmuller, Nam Thoai, Jens Volkert," Error Detection in Large Scale Parallel Programs with Long runtimes, *Future Generation Computer Systems* 19, pp. 689-700, 2003.

[22] 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*, vol. 34, no. 3, pp. 375-408, 2002.

[23] Elnozahy E.N., Johnson D.B. and Zwaenepoel W., "The Performance of Consistent Checkpointing," Proceedings of the 11th Symposium on Reliable Distributed Systems, pp. 39-47, October 1992.

[24] Elnozahy and Zwaenepoel W, " Manetho: Transparent Roll-back Recovery with Low-overhead, Limited Rollback and Fast Output Commit," *IEEE Trans. Computers*, vol. 41, no. 5, pp. 526-531, May 1992.

[25] Elnozahy and Zwaenepoel W, " On the Use and Implementation of Message Logging," *24th int'l Symp. Fault Tolerant Computing*, pp. 298-307, IEEE Computer Society, June 1994.

[26] Hélary J. M., Mostefaoui A. and Raynal M., "Communication-Induced Determination of Consistent Snapshots," *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pp. 208-217, June 1998.

[27] D. Johnson, "Distributed System Fault Tolerance Using Message Logging and Checkpointing," *Ph.D. Thesis, Rice Univ*., Dec. 1989.

[28] JinHo Ahn, Sung-Gi Min, Chong-Sun Hwang, "A Causal Message Logging Protocol for Mobile Nodes in Mobile Computing Environments", *Future Generation Computer Systems* 20, pp 663-686, 2004.

[29] Kalaiselvi, S., Rajaraman, V., "A Survey of Checkpointing Algorithms for Parallel and Distributed Systems", *Sadhna*, Vol. 25, Part 5, October 2000, pp. 489-510.

[30] Kistler, J., and Satyanaranyan, M., " Disconnected Operation in the Coda file system", *ACM Trans. on Computer Systems* 10, 1 (Feb. 1992).

[31] Koo R. and Toueg S., "Checkpointing and Roll-Back Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, vol. 13, no. 1, pp. 23-31, January 1987.

[32] L. Kumar, M. Misra, R.C. Joshi, "Low overhead optimal checkpointing for mobile distributed systems" *Proceedings. 19th IEEE International Conference on Data Engineering*, pp 686 – 88, 2003.

[33] Lalit Kumar, Parveen Kumar, R K Chauhan, "Pitfalls in Minimum-process Coordinated Checkpointing protocols for Mobile Distributed", *ACCST Journal of Research*, Volume III, No. 1, 2005 pp. 51-56.

[34] Lalit Kumar, Parveen Kumar, R K Chauhan, "Message Logging and Checkpointing in Mobile Computing", *Journal of Multi-disciplinary Engineering Technologies*, Vol.1, No.1, 2005, pp. 61-66.

[35] Manivannan D. and Singhal M., "Quasi-Synchronous Checkpointing: Models, Characterization, and Classification," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 7, pp. 703-713, July 1999.

[36] Manivannan D., Netzer R. H. and Singhal M., "Finding Consistent Global Checkpoints in a Distributed Computation," *IEEE Transactions on Parallel & Distributed Systems*, vol. 8, no. 6, pp. 623-627, June 1997.

[37] Netzer, R.H. and Xu,J ,"Necessary and Sufficient Conditions for Consistent Global Snapshots", *IEEE Trans. Parallel and Distributed Systems* 6,2, pp 165-169, 1995.

[38] Parveen Kumar, Lalit Kumar, R K Chauhan, V K Gupta "A Non-Intrusive Minimum Process Synchronous Checkpointing Protocol for Mobile Distributed Systems" *Proceedings of IEEE ICPWC-2005*, January 2005.

[39] Parveen Kumar, Lalit Kumar, R K Chauhan, "A low overhead Non-intrusive Hybrid Synchronous checkpointing protocol for mobile systems", *Journal of Multidisciplinary Engineering Technologies*, Vol.1, No. 1, pp 40-50, 2005.

[40] Parveen Kumar, Lalit Kumar, R K Chauhan, "Synchronous Checkpointing Protocols for Mobile Distributed Systems: A Comparative Study", *International Journal of information and computing science*, Volume 8, No.2, 2005, pp 14-21.

[41] Parveen Kumar, Lalit Kumar, R K Chauhan, "A Hybrid Coordinated Checkpointing Protocol for Mobile Computing Systems", *IETE journal of research*, Vol 52, No. 2&3, pp 247-254, 2006.

[42] Parveen Kumar, Lalit Kumar, R K Chauhan, "A Synchronous Checkpointing Protocol for Mobile Distributed Systems: A Probabilistic Approach, Accepted for Publication in *International Journal of Information and Computer Security*.

[43] Pradhan D.K. and Vaidya N., "Roll-forward Checkpointing Scheme: Concurrent Retry with Non-dedicated Spares," *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 166-174, July 1992.

[44] Pushpendra Singh, Gilbert Cabillic, "A Checkpointing Algorithm for Mobile Computing Environment", *LNCS, No. 2775*, pp 65-74, 2003.

[45] Prakash R. and Singhal M., "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," *IEEE Transaction On Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1035-1048, October1996.

[46] Prakash R. and Singhal M., "Maximum Global Snapshot with Concurrent Initiations", *Proc. Sixth IEEE Symp. Parallel and Distributed Processing*, pp. 344-51, Oct. 1994.

[47] M.L. Powell and D.L. Presotto, "Publishing: A Reliable Broadcase Communication Mechanism", *Proc. ninth Symp. Operating System Principles*, pp. 100-109, ACM SIGOPS, Oct. 1983.

[48] Purnendu Sinha, Da Qi Ren, "Formal Verification of Dependable Distributed Protocols", *Information and Software Technology*, 45, pp. 873-888, 2003.

[49] Quagila, F., Ciciani, R., Baldoni, R., " Checkpointing Protocols in Distributed Systems with Mobile Hosts: A Performance Analysis", *IPPS/SPDP Workshop*, pp. 742-755, 1998.

[50] Randall, B, " System Structure for Software Fault Tolerance", *IEEE Trans. on Software Engineering*, 1,2, 220-232, 1975.

[51] Russell, D.L., "State Restoration in Systems of Communicating Processes", *IEEE Trans. Software Engineering*, 6,2. 183-194, 1980.

[52] R K Chauhan, Parveen Kumar, Lalit Kumar, "A coordinated checkpointing protocol for mobile computing systems", *International Journal of information and computing science*, Accepted for Publication, Vol 9, No. 1, 2006.

[53] R K Chauhan, Parveen Kumar, Lalit Kumar, "Hybrid and intrusive synchronous checkpointing protocols for mobile distributed systems", Accepted for publication in *ACCST Journal of Research*,

Volume IV, No. 4, 2006

[54] R K Chauhan, Parveen Kumar, Lalit Kumar, "Non-intrusive Coordinated Checkpointing Protocols for Mobile Computing Systems : A Critical Survey, *ACCST Journal of Research*, to be published in Volume IV, No. 3, 2006.

[55] R K Chauhan, Parveen Kumar, Lalit Kumar, "Checkpointing Distributed Applications on Mobile Computers", *Journal of Multidisciplinary Engineering and Technologies*, Vol. 2 No.1, Jan. 2006.

[56] Storm R., and Temini, S., "Optimistic Recovery in Distributed Systems", *ACM Trans. Computer Systems*, Aug, 1985, pp. 204-226.

[57] A.P. Sistla and J.L. Welch," Efficient Distributed Recovery Using Message Logging", *Proc. 18th Symp. Principles of Distributed Computing*", pp 223-238, Aug. 1989.

[58] Tamir, Y., Sequin, C.H., "Error Recovery in multi-computers using global checkpoints", *In Proceedings of the International Conference on Parallel Processing*, pp. 32-41, 1984.

[59] S. Venketasan and T.Y. Juang, "Efficient Algorithms for Optimistic Crash recovery", *Distributed Computing*, vol. 8, no. 2, pp. 105-114, June 1994.

[60] S. Venketasan, "Message-Optimal Incremental Snapshots", *Computer and Software Engineering*, vol.1, no.3, pp. 211-231, 1993.

[61] S. Venketasan, " Optimistic Crash recovery Without Rolling back Non-Faulty Processors", *Information Sciences*, 1993.

[62] S. Venketasan and T.T.Y. Juang, "Low Overhead optimistic crash Recovery", *Proc. 11th Int. Conf. Distributed Computing systems*, pp. 454-461, 1991.

[63] Wang Y. M., Huang Y., Vo K.P., Chung P.Y. and Kintala C., "Checkpointing and its Applications," *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*,pp. 22-31, June 1995.

[64] Weigang Ni, Susan V. Vrbsky and Sibabrata Ray, Low-cost Coordinated Checkpointing in Mobile Computing Systems",Proceeding of the Eighth IEEE International Symposium on Computers and Communications, 2003.

[65] Wang Y. and Fuchs, W.K., "Lazy Checkpoint Coordination for Bounding Rollback Propagation," *Proc. 12th Symp. Reliable Distributed Systems*, pp. 78-85, Oct. 1993.

[66] Parveen Kumar, "A Low-Cost Hybrid Coordinated Checkpointing Protocol for Mobile Distributed Systems", *Mobile Information Systems* [An International Journal from IOS Press, Netherlands] pp 13-32, Vol. 4, No. 1, 2007.

[67] Lalit Kumar, Parveen Kumar "A Synchronous Checkpointing Protocol for Mobile Distributed Systems: A Probabilistic Approach", *International Journal of Information and Computer Security* pp 298-314, Vol. 3 No. 1, 2007.

[68] B. Gupta and S. Rahimi "A Novel Low-Overhead Recovery Approach for Distributed Systems", *Journal of Computer Systems, Networks and Communications* pp 1-8, volume 2009.

[69] S. Biswas and S Neogy "A Mobility Based Checkpointing protocol for Mobile Computing System", *International Journal of Computer science and Information Technology*, Vol 2 No. 1, Februrary, 2010.

[70] S Neogy " WTMR – A New Fault Tolerance technique for Wireless and Mobile Computing Systems", *Proceedings of the 11th IEEE International Workshop on Future Trends of Distributed Computing Systems, IEEE*, 2007.

[71] Pourmahmoud, S. Asbaghi, S. Haghighat, A.T. "*23rd International Symposium on Computer and Information Sciences*", 2008.