A Low-Overhead Minimum Process Coordinated Checkpointing Algorithm for Mobile Distributed System

Parveen Kumar¹ Poonam Gahlan²

Abstract-A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. A mobile computing system is a distributed system where some of processes are running on mobile hosts (MHs), whose location in the network changes with time. The number of processes that take checkpoints is minimized to 1) avoid awakening of MHs in doze mode of operation, 2) minimize thrashing of MHs with checkpointing activity, 3) save limited battery life of MHs and low bandwidth of wireless channels. In minimum-process checkpointing protocols, some useless checkpoints are taken or blocking of processes takes place. In this paper, we propose a minimum-process coordinated checkpointing algorithm for non-deterministic mobile distributed systems, where no useless checkpoints are taken. An effort has been made to minimize the blocking of processes and synchronization message overhead. We try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

Keywords-Checkpointing algorithms; parallel & distributed computing; rollback recovery; fault-tolerant system; mobile computing

I. INTRODUCTION

Parallel computing with clusters of workstations 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. It is, thus, necessary to examine both hardware and software solutions to ensure fault tolerance of such parallel computers. 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. It is easier and more cost effective to provide software fault tolerance solutions than hardware solutions to cope with transient failures [25].

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. With the widespread proliferation of the Internet and the emerging global village, the notion of distributed computing systems as a useful and widely deployed tool is becoming a reality [24]. A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features [25]

No common physical clock; This is an important assumption because it introduces the element of “distribution” in the system and gives rise to the inherent asynchrony amongst the processors.

No shared memory; This is a key feature that requires message-passing for communication. It may be noted that a distributed system may still provide the abstraction of a common address space via the distributed shared memory abstraction.

Geographical separation; It is not necessary for the processors to be on a wide-area network (WAN). Recently, the network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system. This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available. The Google search engine is based on the NOW architecture.

Autonomy and heterogeneity; The processors are “loosely coupled” in that they have different speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem [25].

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; i.e., a message whose receive event is recorded, but its send event is lost. To recover from a failure, the system restarts its execution from a previous consistent global state saved on the stable storage during fault-free execution. In distributed systems, checkpointing can be independent, coordinated, or quasi-synchronous. Message Logging is also used for fault tolerance in distributed systems [14]. Most of the existing coordinated checkpointing algorithms [9, 19] rely on the two-phase protocol and save two kinds of checkpoints on the stable

About ¹ Department of Computer Science & Engineering Meerut Institute of Engineering & Technology, Meerut, India, -250005 (e-mail: pk223475@yahoo.com)
About ² Department of Computer Science & Engineering, Singhania University, Pacheri Bari (Rajasthan) India; (e-mail: swastikaarya83@gmail.com)
storage: tentative and permanent. In the first phase, the initiator process takes a tentative checkpoint and requests all or selective processes to take their tentative checkpoints. If all processes are asked to take their checkpoints, it is called all-process coordinated checkpointing [5, 7, 19]. Alternatively, if selective communicating processes are required to take checkpoints, it is called minimum-process checkpointing. Each process informs the initiator whether it succeeded in taking a tentative checkpoint. After the initiator has received positive acknowledgments from all relevant processes, the algorithm enters the second phase. Alternatively, if a process fails to take its tentative checkpoint in the first phase, the initiator process requests all processes to abort their tentative checkpoint.

If the initiator learns that all concerned processes have successfully taken their tentative checkpoints, the algorithm enters the second phase and the initiator asks the relevant processes to make their tentative checkpoints permanent.

In order to record a consistent global checkpoint, when a process takes a checkpoint, it asks (by sending checkpoint requests to) all relevant processes to take checkpoints. Therefore, coordinated checkpointing suffers from high overhead associated with the checkpointing process [20, 21, 22, 23]. Much of the previous work [2, 3, 4, 20, 21, 22, 23] in coordinated checkpointing has focused on minimizing the number of synchronization messages and the number of checkpoints during the checkpointing process. However, some algorithms (called blocking algorithms) force all relevant processes in the system to block their computations during the checkpointing process [3, 9, 21, 22, 23]. Checkpointing includes the time to trace the dependency tree and to save the states of processes on the stable storage, which may be long. Moreover, in mobile computing systems, due to the mobility of MHs, a message may be routed several times before reaching its destination. Therefore, blocking algorithms may dramatically reduce the performance of these systems [7]. Recently, non-blocking algorithms [7, 19] have received considerable attention. In these algorithms, processes need not block during the checkpointing by using a checkpointing sequence number to identify orphan messages. Moreover, these algorithms [4, 10] require all processes in the system to take checkpoints during checkpointing, even though many of them may not be necessary.

A mobile computing system is a distributed system where some of processes are running on mobile hosts (MHs), whose location in the network changes with time. To communicate with MHs, mobile support stations (MSSs) act as access points for the MHs by wireless networks. Features that make traditional checkpointing algorithms for distributed systems unsuitable for mobile computing systems are: locating processes that have to take their checkpoints, energy consumption constraints, lack of stable storage in MHs, and low bandwidth for communication with MHs [1]. Minimum-process coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications, since it avoids domino effect, minimizes the stable storage requirement and also forces only interacting processes to checkpoint.

In coordinated or synchronous checkpointing, processes coordinate their local checkpointing actions such that the set of all recent checkpoints in the system is guaranteed to be consistent [add reference list....]. In case of a fault, every process restarts from its most recent permanentcommitted checkpoint. Hence, this approach simplifies recovery and it does not suffer from domino-effect. Furthermore, coordinated checkpointing requires each process to maintain only one permanent checkpoint on stable storage, reducing storage overhead and eliminating the need for garbage collection. Its main disadvantage is the large latency involved in output commit.

A straightforward approach to coordinate checkpointing is to block communications while the checkpointing process executes. A coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives a message, it stops its execution, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgement message back to the coordinator. After the coordinator receives acknowledgement from all processes, it broadcasts a commit message that completes the two phase checkpointing protocol. After receiving the commit message, each process receives the old permanent checkpoint and makes the tentative checkpoint permanent. The process is then free to resume execution and exchange messages with other processes. The coordinated checkpointing algorithms can also be classified into following two categories: minimum-process and all process algorithms.

Prakash-Singhal algorithm [13] forces only a minimum number of processes to take checkpoints and does not block the underlying computation during checkpointing. However, it was proved that their algorithm may result in an inconsistency [3]. Cao and Singhal [4] achieved non-intrusiveness in the minimum-process algorithm by introducing the concept of mutable checkpoints. The number of useless checkpoints in [4] may be exceedingly high in some situations [16]. Kumar et. al [16] and Kumar et. al [11] reduced the height of the checkpointing tree and the number of useless checkpoints by keeping non-intrusiveness intact, at the extra cost of maintaining and collecting dependency vectors, computing the minimum set and broadcasting the same on the static network along with the checkpoint request. Some minimum-process blocking algorithms are also proposed in literature [3, 9, 21, 23].

In this paper, we propose an efficient checkpointing algorithm for mobile computing systems that forces only a minimum number of processes to take checkpoints. An effort has been made to minimize the blocking of processes and synchronization message overhead.

We capture the partial transitive dependencies during the normal execution by piggybacking dependency vectors onto computation messages. The Z-dependencies are well taken care of in this protocol. In order to reduce the message overhead, we also avoid collecting dependency vectors of all processes to find the minimum set as in [3], [11], [21]. We also try to minimize the loss of checkpointing effort when any process fails to take its checkpoint.
II. PROPOSED CHECKPOINTING ALGORITHM

Our system model is similar to [4, 21]. We propose to handle node mobility and failures during checkpointing as proposed in [21].

A. The Proposed Algorithm

First phase of the algorithm: When a process, say Pi, running on an MH, say MHi, initiates a checkpointing, it sends a checkpoint initiation request to its local MSS, which will be the proxy MSS (if the initiator runs on an MSS, then the MSS is the proxy MSS). The proxy MSS maintains the dependency vector of Pi say Ri. On the basis of Ri, the set of dependent processes of Pi is formed, say Sminset. The proxy MSS broadcasts ckpt (Sminset) to all MSSs. When an MSS receives ckpt (Sminset) message, it checks, if any processes in Sminset are in its cell. If so, the MSS sends mutable checkpoint request message to them. Any process receiving a mutable checkpoint request takes a mutable checkpoint and sends a response to its local MSS. After an MSS received all response messages from the processes to which it sent mutable checkpoint request messages, it sends a response to the proxy MSS. It should be noted that in the first phase, all processes take the mutable checkpoints. For a process running on a static host, mutable checkpoint is equivalent to tentative checkpoint. But, for an MH, mutable checkpoint is different from tentative checkpoint. In order to take a tentative checkpoint, an MH has to record its local state and has to transfer it to its local MSS. But, the mutable checkpoint is stored on the local disk of the MH. It should be noted that the effort of taking a mutable checkpoint is very small as compared to the tentative one[4].

Second Phase of the Algorithm: After the proxy MSS has received the response from every MSS, the algorithm enters the second phase. If the proxy MSS learns that all relevant processes have taken their mutable checkpoints successfully, it asks them to convert their mutable checkpoints into tentative ones and also sends the exact minimum set along with this request. Alternatively, if initiator MSS comes to know that some process has failed to take its checkpoint in the first phase, it issues abort request to all MSS. In this way the MHs need to abort only the mutable checkpoints, and not the tentative ones. In this way we try to reduce the loss of checkpointing effort in case of abort of checkpointing algorithm in first phase.

When an MSS receives the tentative checkpoint request, it asks all the process in the minimum set, which are also running in itself, to convert their mutable checkpoints into tentative ones. When an MSS learns that all relevant process in its cell have taken their tentative checkpoints successfully, it sends response to proxy MSS.

Third Phase of the Algorithm: Finally, when the proxy MSS learns that all processes in the minimum set have taken their tentative checkpoints successfully, it issues commit Sminset, therefore, Pj sends mutable checkpoint request to Pk. Consequently, P3 takes its mutable checkpoint C31. request to all MSSs. When a process in the minimum set gets the commit request, it converts its tentative checkpoint into permanent one and discards its earlier permanent checkpoint, if any.

B. Message Handling During Checkpointing

When a process takes its mutable checkpoint, it does not send any massage till it receives the tentative checkpoint request. Suppose, Pi sends m to Pj after taking its mutable checkpoint and Pj has not taken its mutable checkpoint at the time of receiving m. In this case, if Pj takes its mutable checkpoint after processing m, then m will become orphan. Therefore, we do not allow Pi to send any massage unless and until every process in the minimum set have taken its mutable checkpoint in the first phase. Pi can send massages when it receives the tentative checkpoint request; because, at this moment every concerned process has taken its mutable checkpoint and m cannot become orphan. The massages to be sent are buffered at senders end. In this duration, a process is allowed to continue its normal computations and receive massages.

Suppose, Pj gets the mutable checkpoint request at MSSp. Now, we find any process Pk such that Pk does not belong to Sminset and Pk belongs to Rj. In this case, Pk is also included in the minimum set; and Pj sends mutable checkpoint request to Pk. It should be noted that the Sminset, computed on the basis of dependency vector of initiator process is only a subset of the minimum set. Due to zigzag dependencies, initiator process may be transitively dependent upon some more process which is not included in the Sminset.

C. An Example

The proposed Algorithm can be better understood by the example shown in Figure 2. There are six processes (P0 to P5) denoted by straight lines. Each process is assumed to have initial permanent checkpoints with csn equal to “0”.

Cix denotes the xth checkpoints of Pi. Initial dependency vectors of P0, P1, P2, P3, P4, P5 are [000001], [000010] [000100], [001000], [010000], and [100000], respectively. The dependency vectors are maintained as explained in Section 2.1. P0 sends m2 to P1 along with its dependency vector [000001]. When P1 receives m2, it computes its dependency vector by taking bitwise logical OR of dependency vectors of P0 and P1, which comes out to be [000011]. Similarly, P2 updates its dependency vector on receiving m3 and it comes out to be [000111]. At time t1, P2 initiates checkpointing algorithm with its dependency vector is [000111]. At time t1, P2 finds that it is transitively dependent upon P0 and P1. Therefore, P2 computes the tentative minimum set Sminset = {P0, P1, P2}. P2 sends the mutable checkpoint request to P1 and P0 and takes its own mutable checkpoint C21. For an MH the mutable checkpoint is stored on the disk of MH. It should be noted that Sminset is only a subset of the minimum set.

When P1 takes its mutable checkpoint C21, it finds that it is dependent upon P3 due to m8, but P3 is not a member of Sminset. After taking its mutable checkpoint C21, P2 generates m8 for P3. As P2 has already taken its mutable checkpoint for
the current initiation and it has not received the tentative checkpoint request from the initiator; therefore P2 buffers m8 on its local disk. We define this duration as the uncertainty period of a process during which a process is not allowed to send any massage. The massages generated for sending are buffered at the local disk of the sender’s process. P2 can sends m8 only after getting tentative checkpoint request or abort massages from the initiator process. Similarly, after taking its mutable checkpoint P0 buffers m10 for its uncertainty period. It should be noted that P1 receives m10 only after taking its mutable checkpoint. Similarly, P3 receives m8 only after taking its mutable checkpoint C31. A process receives all the massages during its uncertainty period for example P3 receives m11. A process is also allowed to perform its normal computations during its uncertainty period. At time t2, P2 receives responses to mutable checkpoints requests from all process in the minimum set (not shown in the Figure 2) and finds that they have taken their mutable checkpoints successfully, therefore, P2 issues tentative checkpoint request to all processes. On getting tentative checkpoint request, processes in the minimum set [P0, P1, P2, P3] convert their mutable checkpoints into tentative ones and send the response to initiator process P2; these process also send the massages, buffered at their local disks, to the destination processes. For example, P0 sends m10 to P1 after getting tentative checkpoint request [not shown in the figure]. Similarly, P3 sends m8 to P3 after getting tentative checkpoint request. At time t3, P2 receives responses from the process in minimum set [not shown in the figure] and finds that they have taken their tentative checkpoints successfully, therefore, P2 issues commit request to all process. A process in the minimum set converts its tentative checkpoint into permanent checkpoint and discards its old permanent checkpoint if any.

![Figure 2](image_url)

**D. Correctness Proof**

We can show that the global state collected by the proposed protocol will be consistent. We can prove the result by contradiction. Suppose there is some orphan message in the recorded global state. We explore different possibilities with the help of Figure 2. Suppose, P0 sends m10 after taking its mutable checkpoint and P1 receives m10 before taking its mutable checkpoint. This situation is not possible, because, after taking its mutable checkpoint P0 comes into its uncertainty period and it cannot send any message unless and until it receives the tentative checkpoint request. P2 can issue the tentative checkpoint request only after getting confirmed that every concerned process (including P1) has taken its mutable check point. Hence P1 cannot receive m10 before taking its mutable checkpoint C11. Suppose, P5 sends m13 to P3 after C50 and P3 gets m13 before C31 (not show in the Figure 2). In this case, when P3 takes its mutable checkpoint C31, it will find that P5 does not belong to Sminset and P3 is dependent upon P5; therefore, P3 will send mutable checkpoint request to P5 and send (m13) will also be included in the global state the other possibilities can be proved by obviousness [21].
III. COMPARATIVE ANALYSIS OF THE PROPOSED ALGORITHMS WITH OTHER ALGORITHMS

We use following notations to compare our algorithm with other algorithms:

\(N_{\text{mss}}:\) number of MSSs.

\(N_{\text{min}}:\) number of MHs.

\(C_{\text{pp}}:\) cost of sending a message from one process to another.

\(C_{\text{bb}}:\) cost of sending a message between any two MSSs.

\(C_{\text{mm}}:\) cost of sending a message from an MH to its local MSS (or vice versa).

\(C_{\text{broadcast}}:\) cost of broadcasting a message over static network.

\(C_{\text{search}}:\) cost incurred to locate an MH and forward a message to its current local MSS, from a source MSS.

\(T_{\text{m}}:\) average message delay in static network.

\(T_{\text{wl}}:\) average message delay in the wireless network.

\(T_{\text{ch}}:\) average delay to save a checkpoint on the stable storage. It also includes the time to transfer the checkpoint from an MH to its local MSS.

\(N:\) total number of processes

\(N_{\text{min}}:\) number of minimum processes required to take checkpoints.

\(N_{\text{muc}}:\) number of useless mutable checkpoints [4].

\(T_{\text{search}}:\) average delay incurred to locate an MH and forward a message to its current local MSS.

\(N_{\text{use}}:\) average number of useless checkpoint requests in [4].

\(N_{\text{dep}}:\) average number of processes on which a process depends.

\(h_1:\) height of the checkpointing tree in Koo-Toueg algorithm [9].

\(h_2:\) height of the checkpointing tree in the proposed algorithm.

IV. MESSAGE OVERHEAD OF THE PROPOSED ALGORITHM

A. Message overhead in the first phase

Initiator process sends mutable checkpoint request to the local MSS and (say MSS\(_{\text{mss}}\)) and gets response from the MSS\(_{\text{mss}}\): 2*C\(_{\text{wl}}\)

MSS in broadcasts mutable checkpoint request over the static network: C\(_{\text{broadcast}}\)

We suppose that all the processes are running on MHs.

All the processes in the minimum set get the mutable checkpoint request from the local MSS and sends response to the local MSS: 2*N\(_{\text{min}}\)*C\(_{\text{wl}}\)

Every MSS sends response to MSS\(_{\text{mss}}\): N\(_{\text{min}}\)*C\(_{\text{st}}\)

B. MESSAGE OVERHEAD IN THE SECOND PHASE

MSS\(_{\text{in}}\) broadcasts tentative checkpoint request over static network: C\(_{\text{broadcast}}\)

Every process in the minimum set receives tentative checkpoint request, and sends response to these requests to local MSS: 2*N\(_{\text{min}}\)*C\(_{\text{wl}}\)

Every MSS sends response to MSS\(_{\text{in}}\): N\(_{\text{mss}}\)*C\(_{\text{st}}\)

C. MESSAGE OVERHEAD IN THE THIRD PHASE

MSS\(_{\text{in}}\) broadcasts commit request over static network: C\(_{\text{broadcast}}\)

Total Average message overhead: 2*C\(_{\text{wl}}\)+3*C\(_{\text{broadcast}}\)+4*N\(_{\text{min}}\)*C\(_{\text{wl}}\)+2*N\(_{\text{mss}}\)*C\(_{\text{st}}\)

Our algorithm is a three phase algorithm; therefore it suffers from extra message overhead of C\(_{\text{broadcast}}\)+4*N\(_{\text{min}}\)*C\(_{\text{wl}}\). By doing so, we are able to reduce the loss of checkpointing effort in case of abort of the checkpointing procedure in the first phase. In other algorithms [2, 3, 4, 9], in case of abort in the first phase, all concerned processes are forced to abort their tentative checkpoint whereas in the proposed scheme, all relevant processes abort their mutable checkpoints only. The effort of taking a mutable checkpoint is negligible as compared to tentative one in the mobile distributed system [4]. Frequent abort of checkpointing algorithms, due to exhausted battery power, abrupt disconnections etc., may significantly increase the checkpointing overhead in two-phase algorithms [2, 3, 4, 9]. We try to minimize the same by designing the three phase algorithm.

In our algorithm, only minimum number of processes is required to take their checkpoints.

The blocking time of the Koo-Toueg [11] protocol is highest, followed by Cao-Singhal [4] algorithm. We claim that the blocking time in the proposed scheme will be significantly smaller as compared to the KT Algorithm [9]. Because, in algorithm [9], transitive dependencies are collected by direct dependencies. The checkpoint initiator process, say Pi, sends the checkpoint request to any process Pi if Pi is causally dependent upon Pi. Similarly, Pi sends the checkpoint request to any process Pj if Pi is causally dependent upon Pj. In this way, a checkpointing tree is formed. In the proposed algorithm, transitive dependencies are captured during normal execution as described in Section 2.1. Some zigzag dependencies may not be captured in the proposed scheme during normal execution and they may form low order checkpointing tree in some typical situations. But, in general, the checkpointing tree formed in the proposed scheme will be negligibly small as compared to KT algorithm [9] and hence the blocking time of processes will be small in the proposed scheme as compared to KT algorithm [9]. Furthermore, in the proposed scheme, a checkpoint may be negligibly small as compared to tentative checkpoint. Hence, in the proposed scheme, the blocking period of a process will be significantly small as compared to the KT algorithm [9]. Our blocking period is larger than CS algorithm [3], but it suffers from extra message overhead of collecting dependency vectors from all processes and moreover, it forces all the processes to block for a short duration. In our scheme, a process is blocked only if it is a
member of the minimum set. Furthermore, a process is allowed to perform its normal computations and receive messages during its blocking period.

In the algorithms proposed in [4], [20], no blocking of processes takes place, but some useless checkpoints are taken, which are discarded on commit. In Elnozahy et al [7] algorithm, all processes take checkpoints. In the protocols [3], [9], and in the proposed one, only minimum numbers of processes record their checkpoints. In algorithm [4], concurrent executions of the algorithm are allowed, but it may lead to inconsistencies in doing so [17]. We avoid the concurrent executions of the proposed algorithm.

Table 1. A Comparison of System Performance

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Avg. blocking Time</th>
<th>Average No. of checkpoints</th>
<th>Average Message Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cao-Singhal [4]</td>
<td>$2T_{st}$</td>
<td>$N_{\text{min}}$</td>
<td>$3C_{\text{broadcast}} + 2C_{\text{w}} + 2N_{\text{mss}}C_{\text{st}} + 3N_{\text{mss}}C_{\text{of}}$</td>
</tr>
<tr>
<td>Cao-Singhal [5]</td>
<td>0</td>
<td>$N_{\text{min}} + N_{\text{mut}}$</td>
<td>$2N_{\text{mss}}C_{\text{pp}} + 2N_{\text{mss}}C_{\text{of}}$</td>
</tr>
<tr>
<td>Koo-Toeg Algorithm [11]</td>
<td>$h_1T_{ch}$</td>
<td>$N_{\text{min}}$</td>
<td>$3N_{\text{mss}}C_{\text{pp}} + N_{\text{dep}}$</td>
</tr>
<tr>
<td>Elnozahy et al [8]</td>
<td>0</td>
<td>$N$</td>
<td>$2C_{\text{w}} + 3 + C_{\text{broadcast}} + 4N_{\text{mss}}C_{\text{st}} + 2N_{\text{mss}}C_{\text{of}}$</td>
</tr>
<tr>
<td>Proposed Algorithm</td>
<td>$h_2T_{ch}$</td>
<td>$N_{\text{min}}$</td>
<td>$3N_{\text{mss}}C_{\text{pp}} + 2N_{\text{mss}}C_{\text{of}}$</td>
</tr>
</tbody>
</table>

V. CONCLUSION

In this paper, we have proposed a minimum-process checkpointing protocol for deterministic mobile distributed systems, where no useless checkpoints are taken and an effort has been made to minimize the blocking of processes. We try to reduce the checkpointing time and blocking time of processes by limiting checkpointing tree which may be formed in other algorithms [4, 9]. We captured the transitive dependencies during the normal execution by piggybacking dependency vectors onto computation messages. The Z-dependencies are well taken care of in this protocol. We also try to reduce the loss of checkpointing effort when any process fails to take its checkpoint in coordination with others.

VI. REFERENCES


