2009

A Novel Low-Overhead Recovery Approach for Distributed Systems

B. Gupta
Southern Illinois University Carbondale

Shahram Rahimi
Southern Illinois University Carbondale, rahimi@cs.siu.edu

Published in Gupta, B., & Rahimi, S. (2009). A novel low-overhead recovery approach for distributed systems. Journal of Computer Systems, Networks, and Communications, 2009, Article ID 409873 doi:10.1155/2009/409873

Hindawi Publishing Corporation
Journal of Computer Systems, Networks, and Communications
Volume 2009, Article ID 409873, 8 pages
doi:10.1155/2009/409873

Research Article

A Novel Low-Overhead Recovery Approach for Distributed Systems

B. Gupta and S. Rahimi

Computer Science Department, Southern Illinois University, Carbondale, IL 62901, USA

Correspondence should be addressed to B. Gupta, bidyut@cs.siu.edu

Received 24 November 2008; Accepted 31 March 2009

Recommended by Urs Bapst

We have addressed the complex problem of recovery for concurrent failures in distributed computing environment. We have proposed a new approach in which we 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, we have introduced two new ideas. First, the proposed value of the common checkpointing interval is 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 is always done a priori by an initiator process; besides it is done while the normal distributed application is running. This is quite meaningful because it does not delay the recovery approach in any way.

1. Introduction

It is known that checkpointing and rollback recovery are widely used techniques that allow a distributed computing to progress in spite of a failure [1–11]. A global checkpoint of an n-process distributed system consists of n checkpoints (local) such that each of these n checkpoints corresponds uniquely to one of the n processes. A global checkpoint M is defined as a consistent global checkpoint (state) if no message is sent after a checkpoint of M and received before another checkpoint of M [4]. That is, there must not exist any orphan message between any two local checkpoints belonging to the consistent global checkpoint. The checkpoints belonging to a consistent global checkpoint (state) are called globally consistent checkpoints (GCCs).

There are two fundamental approaches for checkpointing and recovery. One is the asynchronous approach, and the other one is the synchronous approach [12]. In the asynchronous approach, processes take their checkpoints independently. So, taking checkpoints is very simple as there is no coordination needed among the processes while taking the checkpoints. After a failure occurs, a procedure for rollback recovery attempts to build a consistent global checkpoint. However, in this approach because of the absence of any coordination among the processes there may not exist a recent consistent global checkpoint which may cause a rollback of the computation. This is known as domino effect. In the worst case of the domino effect, after the system recovers from a failure all processes may have to rollback to their respective initial states to restart their computation again.

Synchronous checkpointing approach assumes that a single process other than the application processes invokes the checkpointing algorithm periodically to determine a consistent global checkpoint. This process is known as initiator process. It asks periodically all application processes to take checkpoints in a coordinated way. The coordination is done in a way so that the checkpoints taken by the application processes always form a consistent global checkpoint of the system. This coordination is actually achieved through the exchange of additional (control) messages. It causes some delay (known as synchronization delay) during normal operation. This is the main drawback of this method. However, the main advantage is that the set of the checkpoints taken periodically by the different processes always represents a consistent global checkpoint. So, after

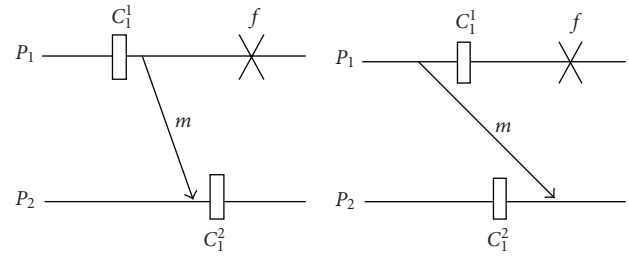
the system recovers from a failure, each process knows where to rollback for restarting its computation again. In fact, the restarting state will always be the most recent consistent global checkpoint. Therefore, recovery is very simple. Hence, compared to the asynchronous approach, taking checkpoints is more complex while recovery is much simpler. Observe that synchronous approach is free from any domino effect. The above discussion is all about determining a recovery line such that there is no orphan message in the distributed system. In this work in addition to orphan messages, we also take care of any lost and delayed messages as well. Before we go further, we have stated briefly why we need to consider these messages.

Consider a simple example of a distributed system with only two processes as shown in Figure 1(a). Process P_1 after taking the checkpoint C_1^1 sends the message m to process P_2. The receiving process P_2 processes the message and then takes its checkpoint C_1^2 and continues. Now assume that a failure f has occurred at process P_1. After the system recovers from the failure, assume that both processes will restart from their respective recent checkpoints C_1^1 and C_1^2. However, process P_1 will resend the message m again since it did not have the chance to record the sending event of the message. Thus process P_2 will receive it again and process it again, even though it did process it once before it took its checkpoint C_1^2. This duplicate processing of the message will result in wrong computation. This message is called an orphan because the receiving event of the message is recorded by the receiving process in its recent local checkpoint C_1^2; where as its sending event is not recorded. Unless proper care is taken, if the processes indeed restart from these two checkpoints, the distributed application will result in wrong computation due to the presence of the orphan message.

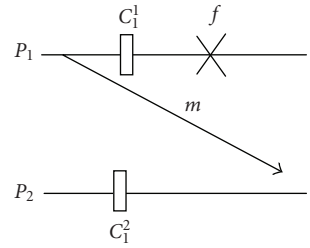
Now consider Figure 1(b). As above assume that after recovery processes restart from their respective checkpoints C_1^1 and C_1^2. Note that the sending event of the message m has already been recorded by the sending process P_1 in its recent checkpoint C_1^1, and so it will not resend it, because it knows that it has already sent the message to P_2. However, the receiving event of the message m has not been recorded by P_2, since it occurred after P_2 took its checkpoint. As a result, P_2 will not get the message again, even though for correct operation it needs the message. In this situation message m is called a lost message. Therefore, for correct operation any such lost message needs to be logged and resent when the system restarts after recovery.

Next consider Figure 1(c). It is seen that because of some reason the message m has been delayed and P_2 did not even receive it before the failure occurred. Now as in the case of the lost message, if the processes restart from their respective checkpoints as shown, process P_1 will not resend it and as a result, process P_2 will not get the message again, even though for correct operation it needs the message. In this situation message m is called a delayed message. Therefore, for correct operation any such delayed message needs to be logged and resent when the system restarts after recovery.

1.1. Problem Formulation. In this paper, we address the following problem: given the recent local checkpoint of each process in a distributed system, after the system recovers from a failure how to handle properly any orphan, lost, or delayed message so that all processes can restart from their respective recent (latest) checkpoints. It also means that a process will need to save only its recent checkpoint. We also handle concurrent process failures, that is, when two or more processes fail concurrently.



(a) Orphan message (b) Lost message



(c) Delayed message

Figure 1

To fulfill our objective, we assume that processes take checkpoints periodically with the same time period to make sure the nonexistence of any orphan message. The proposed checkpointing algorithm is nonblocking. Also it is a single phase one. We also assume that the time between two consecutive invocations of the checkpointing algorithm, T, is larger than the maximum message passing time between any two processes in the system. The importance of this last assumption will be clear when we discuss delayed and lost messages in Section 2.3. The proposed recovery approach needs to consider only lost messages with respect to the recent checkpoints of the processes.

This paper is organized as follows. Section 2 contains the system model and the necessary data structures. In Section 3, we have stated some problem associated with nonblocking approach. In Sections 4 and 5, we have described the checkpointing and recovery approaches along with their respective performances. Section 6 draws the conclusions.

2. Relevant Data Structures and System Model

2.1. System Model. The distributed system has the following characteristics [13]: processes do not share memory and they communicate via messages sent through channels; processes are deterministic and fail stop.

2.2. Relevant Data Structures. The proposed recovery approach needs the following data structures per process for its execution.

Consider a set of n processes $P_1, P_2, \ldots, P_n$ involved in the execution of a distributed algorithm. We assume that application messages are piggybacked with unique sequence numbers, that is, the kth application message will have k as its sequence number. These sequence numbers are used to preserve the total order of the messages received by each process. Process P_i's xth checkpointing interval is the time between its checkpoints C^i_{x-1} and C^i_x and is denoted as $(C^i_x – C^i_{x-1})$. Each process P_i maintains two vectors, each of size n at its xth checkpoint C^i_x; these are a sent vector $V^i_{x(\text{sent})}$ and a received vector $V^i_{x(\text{recv})}$. These vectors are initialized to zero when the system starts. These vectors are stated below.

(i) $V^i_{x(\text{sent})} = [S^{i1}_x, S^{i2}_x, S^{i3}_x, \ldots, S^{\text{in}}_x]$, where S^{ij}_x represents the largest sequence number of all messages sent by process P_i to process P_j in the interval $(C^i_x – C^i_{x-1})$. Note that $S^{ii}_x = 0$.

(ii) $V^i_{x(\text{recv})} = [R^{i1}_x, R^{i2}_x, R^{i3}_x, \ldots, R^{\text{in}}_x]$, where R^{ij}_x represents the largest sequence number of all messages received by P_i from P_j in the checkpointing interval $(C^i_x – C^i_{x-1})$. Also $R^{ii}_x = 0$.

2.3. Delayed Message and Checkpointing Interval. We now state the reason for considering the value of the common checkpointing interval T to be just larger than the maximum message passing time between any two processes of the system. It is known that to take care of the lost and delayed messages the existing idea is message logging. So naturally the question arises for how long a process will go on logging the messages it has sent before a failure (if at all) occurs. We have shown below that because of the above-mentioned value of the common checkpointing interval T, a process P_i needs to save in its recent local checkpoint C^i_x only all the messages it has sent in the recent checkpointing interval $(C^i_x – C^i_{x-1})$. In other words, we are able to use as little information related to the lost and delayed messages as possible for consistent operation after the system restarts.

Consider the situation shown in Figure 2. As before we will explain using a simple system of only two processes, and the observation is true for distributed system of any number of processes as well. Observe that because of our assumed value of T, the duration of the checkpointing interval, any message m sent by process P_i during its checkpointing interval $(C^i_{x-1} – C^i_{x-2})$ always arrives before the recent checkpoint C^j_x of process P_j. Now assume the presence of a failure f as shown in the figure. Also assume that after recovery, the two processes restart from their recent xth checkpoints. Observe that any such message m does not need to be resent as it is processed by the receiving process P_j before its recent checkpoint C^j_x. So it is obvious that such a message m cannot be either a lost or a delayed message. Therefore, there is no need to log such messages by the sender P_i at its recent checkpoint C^i_x. However, messages, such as m' and m'', sent by process P_i in the interval $(C^i_x – C^i_{x-1})$ may be lost or delayed. So in the event of a failure, f, in order to avoid any inconsistency in the computation after the system restarts from the recent checkpoints, we need to log only such sent messages at the recent checkpoint C^i_x of the sender so that they can be resent after the processes restart. Observe that in the event of a failure, any delayed message, such as message m'', is essentially a lost message as well. Hence, in our approach, we consider only the recent checkpoints of the processes and the messages logged at these recent checkpoints are the ones sent only in the recent checkpointing interval. From now on, by "lost message" we will mean both lost and delayed messages. Observe that without such an assumption about the value of the common checkpointing interval T, the messages logged at C^i_x may include not only the ones which a process P_i has sent in its current interval $(C^i_x – C^i_{x-1})$, but also those which P_i sent in the previous intervals as well.

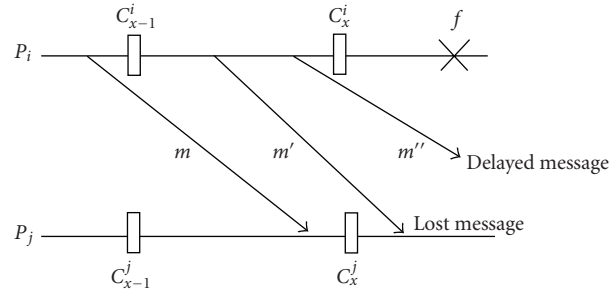


Figure 2: Message m cannot be a delayed or a lost message

Note that in the above discussion, we have implicitly assumed the nonexistence of any abnormally excessive delay in message communication that violates our logical assumption that any message m sent by process P_i during its checkpointing interval $(C^i_{x-1} – C^i_{x-2})$ always arrives before the recent checkpoint C^j_x of process P_j.

3. Problems Associated with Nonblocking Approach

It is known that the classical synchronous checkpointing scheme has three phases: first an initiator process sends a request to all processes to take checkpoints; second the processes take temporary checkpoints and reply back to the initiator process; third the initiator process asks them to convert the temporary checkpoints to permanent ones. Only after that processes can resume their normal computation. In this paper, our objective is to design a single phase nonblocking synchronous approach that guarantees the nonexistence of any orphan message; however it does have some problem. We explain first the problem associated with nonblocking synchronous checkpointing approach. After that we will state a solution. The following discussion although considers only two processes, still the arguments given are valid for any number of processes. Consider a system of two processes P_i and P_j. Assume that the checkpointing algorithm has been initiated by an initiator process P^*, and it has sent a request message M_c to P_i and P_j asking them to take a checkpoint each. In our approach no additional control

message exchange is necessary for making individual recent checkpoints mutually consistent. That is, in this case both processes P_i and P_j will act independently. Let P_i receive the request message M_c and take its checkpoint C_1^i. Let us assume that P_i now immediately sends an application message m to P_j. Suppose at time $(t + €)$, where € is very small with respect to t, P_j receives m. Also suppose that P_j has not yet received M_c from the initiator process. So, P_j has no idea if the checkpointing algorithm has started or not and therefore it processes the message. Now the request message M_c arrives at P_j. Process P_j now takes its checkpoint C_1^j. We find that message m has become an orphan due to the checkpoint C_1^j. Hence, C_1^i and C_1^j cannot be consistent.

To avoid this problem we state a very simple solution. Process P_i piggybacks a flag, say \$, only with its first application message, say m, sent (after it has taken its checkpoint for the current execution of the algorithm and before its next participation in the algorithm) to a process P_j, where $j \neq i$, and $0 \leq j \leq n-1$. Process P_j after receiving the piggybacked application message learns immediately that the checkpointing algorithm has already been invoked; so instead of waiting for the request it takes its checkpoint first, then processes the message m and later it ignores the current request when that arrives.

Note that in our approach an initiator process interacts with the other processes only once via the control message M_c. After receiving M_c, each such process, independent of what others are doing, just takes its checkpoint and sends some vectors to the initiator process and immediately resumes its computation. That is why we consider it as a single-phase algorithm.

4. The Checkpointing Algorithm

Below we describe the nonblocking algorithm. Assume that it is the xth invocation of the algorithm. The algorithm produces n globally consistent checkpoints for a distributed system with n processes; see Algorithm 1.

Proof of Correctness. In the "if" block every process P_i takes its xth checkpoint C_x^i when it receives the request message M_c. That is, none of the messages it has sent before this checkpoint can be an orphan. In the "else" block, a receiving process P_i takes its xth checkpoint C_x^i before processing any application message m, sent by a process which took its xth checkpoint first before sending the message m to P_i. Therefore the message m cannot be an orphan as well. Since this is true for all the processes, hence the recent xth checkpoints C_x^i, $1 \leq i \leq n$ are globally consistent checkpoints.

4.1. Performance. The algorithm is a synchronous one. However it differs from the classical synchronous approach in the following sense; it is just a single phase one unlike the three-phase classical approach, it does not need any exchange of additional (control) messages to coordinate the processes except only the request message M_c, there is no synchronization delay, and finally it is non-blocking. About message complexity the initiator process broadcasts M_c only once and there is one message containing the vectors from each P_i to P^*. So the message complexity is $O(n)$.

4.1.1. Comparisons with Some Existing Works. We use the following notations (and some of the analysis from [10]) to compare our algorithm with some of the most notable algorithms in this area of research, namely, [1, 8, 10]. The analytical comparison is given in Table 1. In this table,

C_{air} is cost of sending a message from one process to another process;

C_{broad} is cost of broadcasting a message to all processes;

$n_{\min}$ is the number of processes that need to take checkpoints;

n is the total number of processes in the system;

n_{dep} is the average number of processes on which a process depends;

T_{ch} is the checkpointing time.

In Table 1, the first column is about blocking. In Koo and Toueg's work, the checkpointing scheme is blocking. So unless all processes take their permanent checkpoints, any underlying distributed application cannot restart. So in the worst case, the total blocking time for the processes that need to take checkpoints is $n_{\min}$ times the checkpointing time T_{ch} per process. For the other works in the table, the algorithms are non-blocking. So they have zero blocking time.

For the second column, consider the work of Cao and Singhal, in the first phase a process uses two system messages while taking a tentative checkpoint. So the system message overhead is $2 * n_{\min} * C_{\text{air}}$. In the second phase the message overhead is $\min(n_{\min} * C_{\text{air}}, C_{\text{broad}})$. So the total overhead is the summation of the above two. In a similar way, the other entries can be explained. Observe that we have a single-phase algorithm, and only one type of system message (a request message) is broadcasted. Therefore C_{broad} is just equal to $n^* C_{\text{air}}$.

Figure 3 illustrates how the number of control messages (system messages) sent and received by processes is affected by the increase in the number of the processes in the system. In Figure 3, n_{dep} factor is considered being 5% of the total number of processes in the system and C_{broad} is equal to $n^* C_{\text{air}}$. We observe that the number of control messages does increase in our approach with the number of processes, but it stays smaller compared to other approaches when the number of the processes is higher than 7 (which is the case most of the time).

5. Recovery Scheme

Our recovery approach is independent of the number of processes that may fail concurrently. In order to identify lost messages in the event of a failure, we adopt only one idea from the centralized approach [14] for message logging: all

At each process P_i $(1 \le i \le n)$
if P_i receives M_c
takes checkpoint C^i_x;
sends its $V^i_{x(\text{sent})}$ and $V^i_{x(\text{recv})}$ to the initiator process P^*;
// all such vectors from each P_i are used by P^* to determine the lost messages
sent by the processes during $(C^i_x – C^i_{x-1})$ in the event of a failure continues its normal operation;
else if P_i receives a piggybacked application message $< m, \$ >$ && P_i has not yet received M_c
for the current execution of the checkpointing algorithm
takes checkpoint C^i_x without waiting for M_c;
sends its $V^i_{x(\text{sent})}$ and $V^i_{x(\text{recv})}$ to the initiator process P^*;
// all such vectors from each P_i are used by P^* to determine the lost messages
sent by the processes during $(C^i_x – C^i_{x-1})$ in the event of a failure continues its normal operation;
// processes the received message m and ignores M_c, when received later

Algorithm 1: Nonblocking Algorithm.

Table 1: System performance.

Algorithm	Blocking time	Messages	Distributed
Koo-Toueg [1]	$n_{\min} * T_{ch}$	$3 * n_{\min} * n_{\text{dep}} * C_{\text{air}}$	Yes
Elnozahy et al. [8]	0	$2 * C_{\text{broad}} + n * C_{\text{air}}$	No
Cao-Singhal [10]	0	$\approx 2 * n_{\min} * C_{\text{air}} + \min(n_{\min} * C_{\text{air}}, C_{\text{broad}})$	Yes
Our algorithm	0	C_{broad}	Yes

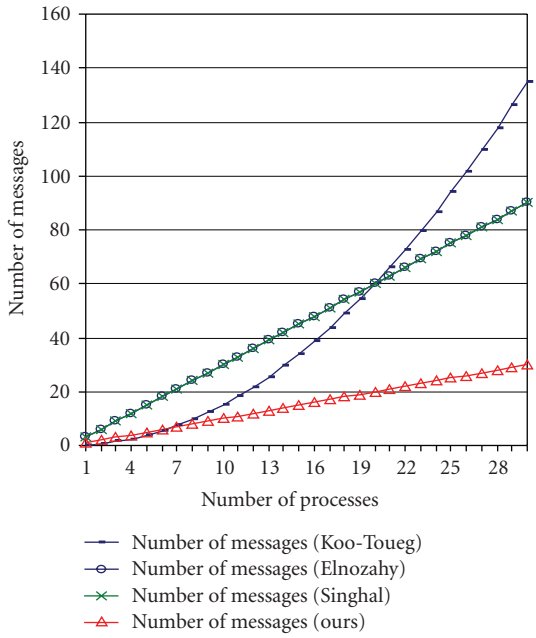


Figure 3: Number of messages versus number of processes for four different approaches

application messages are routed through the initiator process P^*. But, we differ from the centralized approach in that the messages sent to a process P_k are logged at P^* according to the order of their arrival at P^*, and some of these messages may become lost messages in the event of a failure. This is a major difference because the approach in [14] logs copies of only those messages which have been exchanged between any two processes and for doing so it employs an acknowledgment protocol. In our work we denote this message log for process P_k as MESG_k, where $1 \le k \le n$ for an n process distributed system. Another major difference is that in our work the initiator process P^* does not save the checkpoints of the n processes. It is rather the responsibility of the n processes themselves.

The proposed recovery scheme is dependent on the following computation done by the initiator process. Each time when the execution of the checkpointing algorithm is over, the initiator process P^* determines the possible lost messages with respect to the processes' respective recent checkpoints which will be helpful for consistent and correct distributed computation in the event that a failure occurs before the next execution of the checkpointing algorithm. Since this computation can be performed by P^* while the normal distributed application is running, therefore we name it as the Background Computation.

5.1. Background Computation by P.* Assume that the xth execution of the checkpointing algorithm has just been over. So P^* has already collected all the n sent and n received vectors from the n application processes. Using these vectors P^* determines the lost messages, if any, sent by all other processes, P_i $(1 \le i \le n, i \ne k)$ to each P_k in the interval $(C^i_x – C^i_{x-1})$ in the way shown in Algorithm 2.

5.2. Recovery. Let us assume that after the processes have taken their respective xth checkpoints a failure has occurred. It may be concurrent failures also. After the system recovers, initiator process P^* sends to each P_k the lost messages, if any,

Table 2: Brief summary of comparisons.

	Required message ordering	Maximum rollbacks Per failure	Message overhead	Message complexity	Number of concurrent failures
Manivannan-Singhal [7]	None	1	$O(F)$	$O(n^2)$	n
Johnson-Zawenepoel [16]	None	1	$O(1)$	$O(n)$	1
Juang-Venkatesan [19]	None	1	$O(1)$	$O(n^2)$	n
Damini-Grag [17]	None	1	$O(n)$	$O(n^2)$	n
Our algorithm	None	1	None	$O(n)$	n

For each process P_k and $1 \le i \le n,\ i \ne k$
  if $S_x^{ik} > R_x^{ki}$
    P^* records these sequence numbers $(R_x^{ki} + 1)$ to S_x^{ik} in *lost-from-* P_i^k ;
      // messages with sequence numbers $(R_x^{ki} + 1)$ to S_x^{ik} are
      the lost messages from P_i to P_k.
    P^* forms the total order of all lost messages sent by every $P_i,\ i \ne k$ to P_k
  using *lost-from-* P_i^k and the message log MESG_k for P_k;

Algorithm 2

following their total order which P_k did not receive before its recent (xth) checkpoint.

Observe that a failure may occur in the n-process system before the background computation by P^* finishes. Since as in the classical synchronous approach we assume that P^* is not faulty, so P^* will continue with its determination of the lost messages and when it is done it will send these messages to the appropriate receivers.

Theorem 1. *Algorithm nonblocking together with the recovery scheme results in correct computation of the underlying distributed application.*

Proof. According to the checkpointing algorithm there does not exist any orphan message with respect to the recent checkpoints of the processes. Also, the initiator process P^* identifies the lost messages, if any, with respect to the recent local checkpoints of the processes and the recovery approach ensures that the lost messages are resent following their total order to the appropriate destinations after the system restarts. Therefore there does not exist any orphan or lost message with respect to the recent checkpoints. Hence the correctness of the underlying distributed computation is ensured. □

5.3. Performance. The following are the salient features of our approach. First, all processes restart from their respective recent checkpoints; that is, there is no further rollback. It also means that processes save only their recent checkpoints replacing their previous ones. Second, the recovery approach is dependent on the background computation by P^*. This computation goes on in parallel with the normal computation. So it does not delay the recovery approach in any way. It appears to us as a significant advantage. Third, the recovery approach is independent of the number of processes that may fail concurrently. Fourth, the choice of the value of the common checkpointing interval T enables to use as little information related to the lost messages as possible for consistent operation after the system restarts. About the recovered lost messages, it depends on the nature of the distributed application. These messages are computational (application) messages and have to be resent for correct computation. So they do not contribute in any way to the complexity of the recovery approach.

5.3.1. Comparisons with Some Existing Works. In [15], it is a two-phase checkpointing scheme and a process logs both sent and received messages. In our work, it is a single-phase scheme and also only the messages sent are logged. The work in [6] considers only orphan messages, where as our work considers lost and delayed messages as well. However, both the works allow processes to have minimum rollback, thus allowing minimum recomputation.

In the work [7] during normal computation each time a process receives an application message, it has to check if it needs to take a checkpoint so that the received message cannot be an orphan. In our work it is not necessary because of the checkpointing scheme. Hence we avoid some unnecessary comparisons involved in such checking. The message overhead in [7] is $O(F)$, where F is the number of recovery lines established, where as in our work it is absent. Note that by "message overhead" it is meant the size of the control information that is piggybacked with application messages which are exchanged during normal computation. Another important difference is that the work in [7] will establish a recovery line for each failure and then establish a consistent recovery line for the distributed system after the occurrence of concurrent failures. It is not needed in our work, because in our work it does not depend on if it is a single failure or concurrent failures; our recovery line always

consists of the recent checkpoints of the individual processes of the system independent of single or concurrent failures.

When compared to the classical work in [16] the following differences are observed. In [16] there is always an extracontrol messages for each application message, that is, it requires receive sequence number (RSN) and acknowledgment messages in addition to the application message. We do not require it. Besides, the work in [16] has the restriction that during normal computation receiver of a message cannot send a new message until it receives the acknowledgment for the RSN it has sent to the sender of the message which it has already received. This obviously results in slower execution. Our work does not have any restriction of any kind during normal computation. Finally, we handle both single and concurrent failures where as it is only single failure in [16].

The work in [17] employs fault-tolerant vector clock and history mechanism to track causal dependencies, orphan messages, and obsolete messages to bring the system to a consistent state after failures. Our approach is very simple. Our simple checkpointing scheme makes sure that there is no orphan message. Always the consistent state is the set of the recent checkpoints of individual processes. So we do not need any extra effort to determine a consistent state.

The classical work in [18] also employs time stamp vectors to track dependencies in order to determine a consistent state; as mentioned above our approach is always domino-effect free. Also it considers only single failures and its message overhead is $O(n)$. In our work we consider both single and concurrent failures and it does not have any message overhead.

In Table 2 we state a brief summery of comparisons of some important features of some of these checkpointing / recovery approaches.

6. Conclusions

In this work, we have proposed a checkpointing approach that is 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 we 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 T 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 is always done a priori by an initiator process; besides it is done while the normal distributed application is running. It is meaningful because it does not delay the recovery approach in any way. Besides, the recovery approach is independent of the number of processes that may fail concurrently. Finally note that our 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 we consider only processes' recent checkpoints irrespective of their sequence numbers.

References

[1] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 23–31, 1987.

[2] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent global checkpoints based on direct dependency tracking," *Information Processing Letters*, vol. 50, no. 4, pp. 223–230, 1994.

[3] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[4] Y.-M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 456–468, 1997.

[5] B. Gupta, S. K. Banerjee, and B. Liu, "Design of new roll-forward recovery approach for distributed systems," *IEE Proceedings: Computers & Digital Techniques*, vol. 149, no. 3, pp. 105–112, 2002.

[6] B. Gupta, S. Rahimi, and Z. Liu, "Novel low-overhead roll-forward recovery scheme for distributed systems," *IET Computers & Digital Techniques*, vol. 1, no. 4, pp. 397–404, 2007.

[7] D. Manivannan and M. Singhal, "Asynchronous recovery without using vector timestamps," *Journal of Parallel and Distributed Computing*, vol. 62, no. 12, pp. 1695–1728, 2002.

[8] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Proceedings of the 11th Symposium on Reliable Distributed Systems (RELDIS '92)*, pp. 86–95, Houston, Tex, USA, October 1992.

[9] G. Cao and M. Singhal, "On coordinated checkpointing in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1213–1225, 1998.

[10] G. Cao and M. Singhal, "Mutable checkpoints: a new checkpointing approach for mobile computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 2, pp. 157–172, 2001.

[11] S. Venkatesan, T. T.-Y. Juang, and S. Alagar, "Optimistic crash recovery without changing application messages," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 263–271, 1997.

[12] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw-Hill, New York, NY, USA, 1994.

[13] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice-Hall, Upper Saddle River, NJ, USA, 1994.

[14] M. L. Powell and D. L. Presotto, "Publishing: a reliable broadcast communication mechanism," in *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP '83)*, pp. 100–109, Bretton Woods, NH, USA, October 1983.

[15] Q. Jiang, Y. Luo, and D. Manivannan, "An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 12, pp. 1575–1589, 2008.

[16] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *Proceedings of the 17th International Symposium on Fault-Tolerant Computing (FTCS '87)*, pp. 14–19, Pittsburgh, Pa, USA, July 1987.

[17] O. P. Damani and V. K. Garg, "How to recover efficiently and asynchronously when optimism fails," in *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pp. 108–115, Hong Kong, May 1996.

[18] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204–226, 1985.

[19] T.-Y. Juang and S. Venkatesan, "Efficient algorithm for crash recovery in distributed systems," in *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '90)*, pp. 349–361, Bangalore, India, December 1990.