Novel Crash Recovery Approach for Concurrent Failures in Cluster Federation

Bidyut Gupta and Shahram Rahimi

Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
{bidyut, rahimi}@cs.siu.edu

Abstract. In this paper, we have proposed a simple and efficient approach for check pointing and recovery in cluster computing environment. The recovery scheme deals with both orphan and lost intra and inter cluster messages. This check pointing scheme ensures that after the system recovers from failures, all processes in different clusters can restart from their respective recent check-points; thus avoiding any domino effect. That is, the recent check points always form a consistent recovery line of the cluster federation. The main features of our work are: it uses selective message logging which enables the initiator process in each cluster to log the minimum number of messages, the recovery scheme is domino effect free and is executed simultaneously by all clusters in the cluster federation, it considers concurrent failures, message complexities in each cluster for both check pointing and recovery schemes are just O(n), where n is the number of processes in a cluster. These features make our algorithm superior to the existing works.

1 Introduction

Cluster federation is a union of clusters, where each cluster contains a certain number of processes. A Cluster may be defined as an independent computer combined into a unified system through software and networking. Cluster computing environments have provided a cost-effective solution to many distributed computing problems by investing inexpensive hardware [2], [3], [15]. With the growing importance of cluster computing, its fault-tolerant aspect deserves significant attention. It is known that check pointing and rollback recovery are widely used techniques that allows a system to progress in spite of a failure [1]. The basic idea is to periodically record the system state as a checkpoint during normal system operation and upon detection of faults, to restore one of the checkpoints and restart the system from there [4]-[7], [10]-[12], [16]. It may be noted that a distributed system / cluster federation is said to be consistent, if there is no message which is recorded in the state of its receiver but not recorded in the state of its sender [1]-[7]. But if such a message exists, then it is known as orphan message. Such a consistent state of the system is also referred to as a recovery line which in effect consists of one checkpoint per process of the system. It is the responsibility of each cluster to determine its consistent checkpoint set that consists of

N. Abdennadher and D. Petcu (Eds.): GPC 2009, LNCS 5529, pp. 434–445, 2009.
© Springer-Verlag Berlin Heidelberg 2009

one checkpoint from each process present in it. But this consistent checkpoint set of one cluster may not be consistent with the other clusters' consistent checkpoint sets, because clusters interact through inter cluster messages which may result in dependencies among the clusters, meaning thereby that some such inter cluster messages may become orphan messages. Therefore, a collection of consistent checkpoint sets, one from each cluster in the federation, does not necessarily produce a consistent federation level checkpoint (also known as federation level recovery line). Consequently, rollback of one failed process in a cluster may force some other processes in the other clusters to rollback in order to maintain consistency of operation by the cluster federation. In the worst case, consistency requirement may force the system to rollback to the initial state of the system, losing all the work performed before a failure. This uncontrolled propagation of rollback is known as domino-effect [1]. Besides, for correct computation of the underlying distributed computation after the system recovers from failures, a recovery scheme must ensure that all lost messages are identified and replayed to the appropriate processes when they restart.

Problem Formulation: The main objective of this work is three fold. First, it must take care of both orphan messages during the check pointing phase itself unlike the existing works [2],[3],[13],[14]. For this purpose we will consider designing a single phase non blocking check pointing scheme that must take care of both intra and inter cluster orphan messages. Second, it must identify all intra and inter cluster lost messages in an efficient way at the time of recovery. Third, recovery schemes in the different clusters must have to be executed simultaneously and the processes must restart from their respective recent (latest) checkpoints, thereby avoiding the domino effect.

This paper is organized as follows. In Section 2 we have presented the different data structures. In Section 3 we have presented the check pointing algorithm and its performance. Section 4 contains the recovery scheme along with its performance. Section 5 draws the conclusion.

2 Relevant Data Structures and System Model

2.1 System Model

We assume that processes are deterministic in the sense that from the same state, if given the same inputs, a process executes the same sequence of instructions. We also assume that processes are fail stop. It means that upon failure, a process does not perform any incorrect actions and simply ceases to function.

2.2 Notations and Relevant Data Structures

The proposed recovery approach needs the following data structures to be maintained in each cluster.

The k^{th} cluster of the cluster federation is denoted as C^K. The i^{th} process in C^K is denoted as P_i^K. The x^{th} checkpoint taken by the i^{th} process P_i^K in the k^{th} cluster is denoted as $CP^{i,K}{}_x$. An intra cluster message from P_i^K is denoted as m_s^i, where s is the message sequence number assigned by the sender P_i^K. We term this sequence number as the primary sequence number (PSN). An inter cluster message from the i^{th} process

P_i^K (ϵC^K) is denoted as $M_s^{K(i)}$, where s is the PSN of the message. We assume that every cluster has an initiator process which has a two-fold responsibility; first, it is responsible for invoking the check pointing algorithm in this cluster and second, it determines the lost messages in this cluster in the event of failures. For the k^{th} cluster P^k represents the initiator process. In order to identify lost messages in the event of a failure, we assume that for every cluster all intra and inter cluster messages are routed through its initiator process on their way to the respective destinations. Thus in a cluster C^K any message communication between any two application processes takes place via the initiator P^K. To every message (including both intra and inter cluster ones) to be delivered to a process, say P_i^K, the initiator process P^K assigns a new sequence number, termed as the secondary sequence number (SSN) following its order of arrival at the initiator. Each such message is then delivered along with its SSN to the destination process P_i^K. This destination process in its recent checkpoint just remembers only the maximum SSN in $SEQ_{i(max)}$. Note that these SSNs (in ascending order) actually create the total order of the messages sent to a receiving process. In each cluster its initiator process maintains a message log for each process belonging to this cluster. Thus in cluster C^K the initiator process P^K maintains a message log, $MESG\text{-}LOG_i$ for each process P_i^K. This log stores copies of the messages to be delivered to the i^{th} process following their order of arrival at the initiator. P^K saves the maximum SSN found in the message log for process P_i^K in $SSN_{i(max)}$. As an example, assume that P^K has received first an intra cluster message m_2^r, followed by another one, m_6^t from the r^{th} and t^{th} processes respectively for the destination P_i^K. After that it receives an inter cluster message $M_4^{Q(n)}$ coming from the n^{th} process P_n^Q of the q^{th} cluster C^Q for the same destination P_i^K. Note that the three PSNs of the messages are 2, 6, and 4 respectively. However the initiator process P^K now assigns the secondary sequence numbers 1, 2, and 3 to these messages following their order of arrival at it. Now $SSN_{i(max)}$ contains 3. The message log for process P_i^K is stated below along with the messages' respective SSNs appearing in brackets.

$$MESG\text{-}LOG_i^K = [m_2^r\ (1),\ m_6^t\ (2),\ M_4^{Q(n)}\ (3)]$$

2.3 Check Pointing Interval and Selective Message Logging

As in [14], we assume that the value of the common check pointing interval T used in all the clusters is just larger than the maximum message (considering both intra and inter cluster messages) passing time between any two processes of the cluster federation. In the following discussion we have used some of the idea reported in [14]. We now state the benefits for such an assumption. It is known that message logging [17] is used to take care of lost and delayed messages. 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 check pointing interval T, a process P_i^K in cluster C^K needs to save in its recent checkpoint $CP^{i,K}{}_x$ only all the messages it has sent in the recent check pointing interval $(CP^{i,K}{}_x - CP^{i,K}{}_{x-1})$. 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 Fig. 1. For simplicity we will explain using a single cluster, say C^K with only two processes and with intra cluster messages only. Let

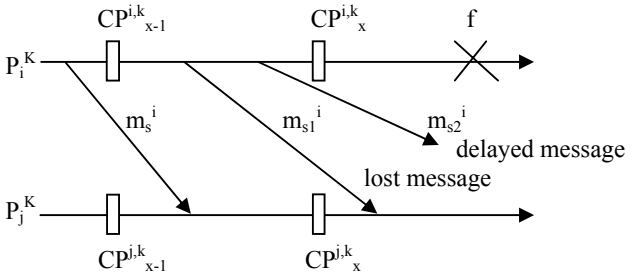

Fig. 1. Messages m_{s1}^i and m_{s2}^i are lost messages

the processes be P_i^K and P_j^K. The observation is true for clusters consisting of any number of processes as well as for inter cluster messages as well. Observe that because of our assumed value of T, the duration of the check pointing interval, any message m_s^i sent by process P_i^K during its check pointing interval ($CP^{i,K}_{x-1} - CP^{i,K}_{x-2}$) always arrives before the recent checkpoint $CP^{j,K}_x$ of process P_j^K. 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 x^{th} checkpoints. Observe that any such message m_s^i does not need to be resent as it is processed by the receiving process P_j^K before its recent checkpoint $C^{j,K}_x$. So it is obvious that such a message m_s^i can not be either a lost or a delayed message. Therefore, there is no need to log such messages by the initiator process P^K. However, messages, such as m_{s1}^i and m_{s2}^i, sent by process P_i^K in the interval ($CP^{i,K}_x - CP^{i,K}_{x-1}$) 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 initiator 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_{s2}^i, 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 the initiator process are the ones sent only in the recent check pointing interval. From now on, by 'lost message' we will mean both lost and delayed message. Observe that without such an assumption about the value of the common check pointing interval T, the above mentioned selective message logging is not possible; rather without such an assumption the messages logged may include not only the ones which a process P_i^K has sent in its current interval ($CP^{i,K}_x - CP^{i,K}_{x-1}$), but also those which P_i^K sent in the previous intervals as well.

3 The Check Pointing Algorithm

It is known that the classical synchronous check pointing scheme for distributed systems has three phases: first an initiator process sends a request to all processes to take checkpoints; second the processes take temporary check points and reply back to the initiator process; third the initiator process asks them to convert the temporary check points to permanent ones. Only after that processes can resume their normal computation. In between every two consecutive phases processes remain blocked. In this work

our objective is to design a single phase non-blocking synchronous approach as in [12] in each cluster. The proposed check pointing algorithm in every cluster works in the following way. Without any loss of generality let us consider a cluster C^K. The algorithm is invoked periodically by the initiator process P^K. In each invocation the initiator sends a request message M_c to the different processes of C^K asking them to take a checkpoint each. Each process after receiving the request message M_c will take its checkpoint independent of what others are doing. As in [12], no additional control message exchange is necessary unlike the classical synchronous approach for making individual recent checkpoints mutually consistent. However the present approach faces similar problem as in [12] regarding making checkpoints consistent. We explain first the problem considering the cluster C^K only and then we will state a solution which is similarly applicable to other clusters as well. Assume that the check pointing algorithm has been initiated by the initiator process P^K and it has sent a request message M_c to the processes of the cluster C^K. Also assume that it is the x^{th} execution of the algorithm. Suppose that after receiving M_c the i^{th} process P_i^K takes its x^{th} checkpoint $CP^{i,K}{}_x$ and immediately then sends an intra cluster application message m_s^i to the j^{th} process P_j^K. Note that in our approach all processes act independently after receiving M_c. Suppose at time $(t + €)$, where € is very small with respect to t, process P_j^K receives the message m_s^i. Also suppose that P_j^K has not yet received M_c from the initiator process. So, it processes the message. Now the request message M_c arrives at P_j^K. Process P_j^K now takes its checkpoint $CP^{j,K}{}_x$. We find that message m_s^i has become an orphan due to the checkpoint $CP^{j,K}{}_x$. Hence, the checkpoints $CP^{i,K}{}_x$ and $CP^{j,K}{}_x$ cannot be consistent.

To avoid this problem we propose the following simple solution. Every sending process P_i^K piggybacks a flag, say \$, only with its first application message, say m_s^i, sent (after it has taken its checkpoint for the current execution of the check pointing algorithm and before its next participation in the algorithm) to any other process P_j^K in the cluster. Process P_j^K after receiving the piggybacked application message learns immediately that the check pointing algorithm has already been invoked; so instead of waiting for the request it takes its checkpoint first, then processes the message m_s^i and later it ignores the current request when that arrives. Observe that the above solution holds good for all inter cluster messages also. The reason is that the x^{th} execution of the check pointing algorithm takes place simultaneously in the different clusters. Note that in our check pointing approach each initiator process interacts with the other processes in its cluster 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 resumes normal computation. That is why we term it as a single phase non-blocking algorithm. Below we describe the algorithm. Assume that it is the x^{th} invocation of the check pointing algorithm.

3.1 Algorithm Non-blocking

For each cluster C^K

At each process P_i^K ($\in C^K$)

if P_i^K receives M_c
takes checkpoint $CP^{i,K}{}_x$;
continues its normal operation;

else if P_i^K receives a piggybacked application message < m_s^j/ $M_s^{Q(t)}$, \$> && P_i^K has not yet received M_c for the current execution of the check pointing algorithm, it takes checkpoint $CP^{i,K}{}_x$ without waiting for M_c; continues its normal operation;

// processes the received intra cluster message m_s^i / inter cluster message $M_s^{Q(t)}$ and ignores M_c, when received later

Proof of Correctness: In the 'if' block every process P_i^K takes its x^{th} checkpoint $CP^{i,K}{}_x$ when it receives the request message M_c. That is, none of the intra / inter cluster messages it has sent before this checkpoint can be an orphan. In the 'else if' block, a receiving process P_i^K takes its x^{th} checkpoint $CP^{i,K}{}_x$ before processing any intra / inter cluster application message m_s^j/ $M_s^{Q(t)}$, sent by a process which took its x^{th} checkpoint first before sending the message to P_i^K. Therefore the message m_s^i/ $M_s^{Q(t)}$ can not be an orphan as well. Since this is true for all the processes, hence all recent x^{th} checkpoints in cluster C^K are mutually consistent. •

Theorem 1. The x^{th} checkpoints of all processes in the cluster federation are mutually consistent.

Proof. Without any loss of generality let us consider two clusters C^K and C^L and we assume that it is the x^{th} invocation of the check pointing algorithm. Observe that the same check pointing interval is used by the respective initiator processes P^K and P^L in these two clusters. Suppose that the i^{th} process P_i^K ($\in C^K$) has just taken its x^{th} checkpoint $CP^{i,K}{}_x$ and immediately after that it sends an inter cluster application message $M_s^{K(i)}$ to the j^{th} process P_j^L ($\in C^L$). According to our proposed solution this message is piggybacked with the flag \$.

Now assume that process P_j^L receives this application message before it receives the request message M_c corresponding to the x^{th} execution of the algorithm from its initiator P^L. If process P_j^L has not yet received any other piggybacked application message yet, whether it is intra or inter cluster, then instead of waiting for the request to come from its initiator P^L, it first takes its x^{th} checkpoint $CP^{j,L}{}_x$, then processes the message $M_s^{K(i)}$ and later it ignores the current request when that arrives. Now observe that the message $M_s^{K(i)}$ cannot be an orphan. Hence the two checkpoints $CP^{i,K}{}_x$ and $CP^{j,L}{}_x$ are mutually consistent. Now assume that process P_j^L receives this piggybacked application message after it receives the request message M_c corresponding to the x^{th} execution of the algorithm from its initiator P^L. This means that process P_j^L has received the inter cluster message after taking its x^{th} checkpoint $CP^{j,L}{}_x$. So obviously the message $M_s^{K(i)}$ cannot be an orphan and hence the two checkpoints $CP^{i,K}{}_x$ and $CP^{j,L}{}_x$ are mutually consistent.

Since the above observation is true for any two checkpoints belonging to different clusters in the cluster federation and also Algorithm Non-blocking guarantees that the checkpoints inside a cluster are mutually consistent, therefore all checkpoints in the cluster federation are mutually consistent. •

3.2 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 except only the request message M_c, there is no synchronization delay, and finally it is non-blocking. However it enjoys the main advantage of the three phase classical approach in that the recent checkpoints are always consistent; so processes after recovery from failures can restart from these checkpoints (i.e. domino-effect free recovery). Therefore, it offers the advantages of both synchronous and asynchronous check pointing approach while avoiding their main drawbacks, such as blocking, synchronization delay, and domino effect. About message complexity in each cluster the initiator process broadcasts M_c only once. So the message complexity is $O(n)$ for an n-process cluster. Also note that it is simultaneously executed in all the clusters.

Since a cluster is nothing but an individual distributed system, so we compare the proposed algorithm used in each cluster with some noted check pointing algorithms.

Comparisons with Some Existing Works. We use the following notations (and some of the analysis from [7]) to compare our algorithm with some of the most notable check pointing algorithms [1], [6], and [7]. The analytical comparison is given in Table 1. In this Table:

C_{air} is average cost of sending a message from one process to another process;
C_{broad} is cost of broadcasting a message to all processes; Note that we assume IP broadcasting.
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 check pointing time;

Table 1. System Performance

Algorithm	*Blocking time*	*Messages*	*Distributed*
Alg. [1]	$n_{min} * T_{ch}$	$3 * n_{min} * n_{dep} * C_{air}$	Yes
Alg. [6]	0	$2 * C_{broad} + n * C_{air}$	No
Alg. [7]	0	$\approx 2 * n_{min} * C_{air} + \min(n_{min} * C_{air}, C_{broad})$	Yes
Our Alg.	0	C_{broad}	Yes

Fig. 2 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 distributed system (cluster). In Fig. 2, the n_{dep} factor is considered being 5% of the total number of processes in the system and C_{broad} is equal to $n*C_{air}$. 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).

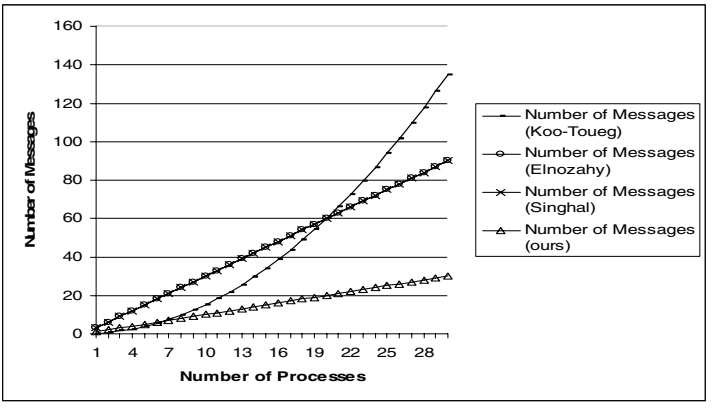

Fig. 2. Number of messages vs. number of processes for four different approaches

4 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 assume that for every cluster all intra and inter cluster messages are routed through the initiator process (s) on their way to their respective destinations. Also, in each cluster, say C^K, the messages sent to the i^{th} process P_i^K in the cluster are logged at its initiator process P^K according to the order of their arrival at the initiator. The message log for P_i^K is denoted as $MESG\text{-}LOG_i^K$.

4.1 Algorithm Recovery

The following recovery algorithm is executed simultaneously in all clusters. It works for any number of concurrent failures.

For each cluster C^K

At each process P_i^K ($\in C^K$):

P_i^K sends its $SEQ_{i(max)}$ to P^K;

At the initiator process P^K:

if $SSN_{i(max)} > SEQ_{i(max)}$

P^K replays to P_i^K the messages with sequence numbers from $SEQ_{i(max)} + 1$ to $SSN_{i(max)}$;

// Lost messages are resent to P_i^K following their total order

P_i^K restarts computation;

else P_i^K restarts computation;

// no lost message to P_i^K exists

Theorem 2. Algorithm Non-blocking together with the recovery scheme results in correct computation of the underlying distributed application.

Proof: According to the check pointing algorithm and Theorem 1 there does not exist any orphan message with respect to the recent checkpoints of the processes in the cluster federation. Also, in each cluster C^K, its initiator process P^K identifies the lost messages, if any, with respect to the recent local checkpoints of the processes in the cluster and the recovery approach ensures through the use of the secondary sequence numbers that the lost messages are resent following their total order to the appropriate destinations in the cluster after the system restarts. Therefore there does not exist any orphan or lost message with respect to the recent checkpoints of the processes in the cluster federation. Hence the correctness of the underlying distributed computation is ensured. ●

4.2 Performance

The following are the salient features of our approach. First of all, processes restart from their respective recent checkpoints; that is there is no further rollback (i.e. domino effect free recovery). It also means that processes save only their recent checkpoints replacing their previous ones. Second, the choice of the value of the common check pointing interval T enables to use as little information related to the lost messages as possible for consistent operation after the system restarts. Third, our work is independent of, if it is a single failure or concurrent failures. Fourth, the recovery approach needs just one control message from each of the processes of a cluster, which carries the $SEQ_{(max)}$. Therefore it needs only n control messages for an n-process cluster and so the message complexity is $O(n)$ in each cluster. Besides, it takes care of both orphan and lost messages. Finally, the recovery scheme is executed simultaneously by all clusters in the cluster federation.

4.3 Comparison

Since a cluster is nothing but an individual distributed system, so first we compare the proposed recovery scheme used in each cluster with some noted recovery algorithms existing in the area of distributed computing.

Comparison with noted recovery approaches in distributed systems: In [5] the message overhead 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 each application message which are exchanged during normal computation. Another important difference is that the work in [5] 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. In the classical work reported in [8] there is always an extra control message for each application message, i.e. it requires receive sequence number (RSN) and acknowledgement messages in addition to the application message. We don't require it. Besides, we handle both single and concurrent failures where as it is only single failures

Table 2. Brief Summary of Comparisons

Algorithm	*Required Message ordering*	*Maximum rollbacks Per failure*	*Message Overhead*	*Message Complexity*	*Number of concurrent Failures*
[5]	None	1	O(F)	$O(n^2)$	n
[8]	None	1	O(1)	O(n)	1
[11]	None	1	O(1)	$O(n^2)$	n
[9]	None	1	O(n)	$O(n^2)$	n
Our Alg.	None	DEF	None	O(n)	n

in [8]. Below in Table 2 we state a brief summery of comparisons of some important features of the different check pointing / recovery approaches. Note that 'DEF' in the 3rd column denotes 'domino effect free'. In the proposed solution of [9], fault-tolerant vector clock has been used to track causal dependencies in spite of failures. In order to determine a consistent state after failure, it uses checkpointing along with a history mechanism that helps to detect orphan and obsolete messages. In [11], an optimistic recovery algorithm has been proposed that uses $O(n^2)$ messages in arbitrary networks. Each application message is appended with information of size O(1).

Comparisons with recovery approaches in cluster federation: We will now compare our approach with some existing works that deal with recovery in cluster federations. The work in [2] has considered a very restricted architecture in which multiple coordinated checkpointing subsystems are connected with a single independent checkpointing subsystem and the multiple coordinated subsystems can not communicate directly with each other; rather they do it via the independent subsystem. The assumed restricted architecture is the main short coming of this work. In the proposed solution of [3], whenever a cluster fails, it broadcasts an alert message after recovering. Each time there is a rollback, the rolled back cluster further broadcasts the alert message triggering the next iteration of the algorithm. The algorithm terminates when there is no more any alert message. The main drawbacks of the algorithm are: it suffers from domino effect; if all clusters have to roll back except the failed cluster, then it may result in a message storm of the alert messages; it does not consider lost messages and concurrent failures. We have none of these drawbacks. The work in [13] considers communication-induced check pointing scheme. Although it does not suffer from any message storm unlike in [3] and it has a better message complexity than in [3], however, it also suffers from domino effect and it considers only single failures and orphan messages. We don't have any such drawback. The work in [14] offers advantages similar to ours. However, its message complexity is $O(N^2)$, where N is the total number of processes in the cluster federation; where as in our work it is O(N) because the message complexity in an n-process cluster is just O(n). Also in [14] authors have used quite complex data structures, for example, each process maintains two vectors of size equal to the number of clusters to handle inter cluster lost messages. Also to handle intra cluster lost messages each process in a cluster maintains two more vectors of size equal to the number of processes inside the cluster. In our work each process maintains just one simple data structure SEQ(max) to save the maximum SSN. In Table 3 we have summarized the comparisons.

Table 3. Comparison Summary. N = Number of processes in the cluster federation, N_1 = Number of clusters in the cluster federation, and K = Number of iterations of the recovery algorithm.

Criteria	*Our Approach*	*Alg [14]*	*Alg [3]*	*Alg [13]*
Architecture Dependent	Yes	No	No	No
Domino – Effect Free	Yes	Yes	No	No
Concurrent Failures	Yes	Yes	No	No
Inter – Cluster Lost Messages	Yes	Yes	No	No
No. of checkpoints / process	1	1	>> 1	>> 1
Message complexity	O (N)	O (N^2)	O (KN_1)	O (KN_1^2)

5 Conclusion

In this work, we have proposed a single phase non-blocking check pointing approach free from any synchronization delay and domino effect. The proposed value of the common check pointing interval T enables to use as little information related to the lost messages as possible for consistent operation. Also, it is independent of the number of processes that may fail concurrently. The message complexity of the check pointing algorithm as well as the recovery approach is just O(n) for an n process cluster. Both the check pointing and recovery schemes are executed simultaneously in all clusters. The proposed schemes are independent of the effect of any clock drift on the respective sequence numbers of the recent checkpoints of the processes, Finally, it should be noted that since existing tools are not sufficient to implement the algorithm, a large amount of additional work is required for its implementation.

References

1. Koo, R., Toueg, S.: Checkpointing and Rollback-Recovery for Distributed Systems. IEEE Trans. Software Engineering, SE-13(1) 1, 23–31 (1987)
2. Cao, J., Chen, Y., Zhang, K., He, Y.: Checkpointing in Hybrid Distributed Systems. In: Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN 2004), Hong Kong, China, pp. 136–141 (2004)
3. Monnet, S., Morin, C., Badrinath, R.: Hybrid Checkpointing for Parallel Applications in cluster Federations. In: Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, IL, USA, pp. 773–782 (2004)
4. Gupta, B., Rahimi, S., Liu, Z.: A Novel Low-Overhead Roll-Forward Recovery Scheme for Distributed Systems. IET Computers and Digital Techniques 1(4), 397–404 (2007)
5. Manivannan, D., Singhal, M.: Asynchronous Recovery without using vector timestamps. Journal of Parallel and Distributed Computing 62, 1695–1728 (2002)
6. Elnozahy, E.N., Johnson, D.B., Zwaenepoel, W.: The Performance of Consistent Check pointing. In: Proceedings of the 11th Symp. Reliable Distributed Systems, pp. 86–95 (1992)
7. Cao, G., Singhal, M.: Mutable Checkpoints. A New Checkpointing Approach for Mobile Computing Systems. IEEE Transactions on Parallel and Distributed Systems 12(2), 157–172 (2001)

8. Johnson, D.B., Zwaenepoel, W.: Sender-Based Message Logging. In: Proceedings of the 17th Fault-Tolerant Computing Symposium, Pittsburgh, pp. 14–19 (1987)
9. Damini, O.P., Garg, V.K.: How to Recover Efficiently and Asynchronously When Optimism Fails. In: Proceedings of the 16th International Conference on Distributed Computing Systems, pp. 108–115 (1996)
10. Venkatesan, S., Juang, T., Alagar, S.: Optimistic Crash Recovery Without Changing Application Messages. IEEE Trans. Parallel and Distributed Systems 8(3), 263–271 (1997)
11. Juang, T., Venkatesan, S.: Efficient Algorithm for Crash Recovery in Distributed Systems. In: Proceedings of the 10th Conference on Foundations on Software Technology and Theoretical Computer Science, pp. 349–361 (1990)
12. Gupta, B., Rahimi, S., Rias, R.A., Bangalore, G.: A Low-Overhead Non-Blocking Checkpointing Algorithm for Mobile Computing Environment. In: Chung, Y.-C., Moreira, J.E. (eds.) GPC 2006. LNCS, vol. 3947, pp. 597–608. Springer, Heidelberg (2006)
13. Gupta, B., Rahimi, S., Ahmad, R., Chirra, R.: A Novel Recovery approach for Cluster Federations. In: Cérin, C., Li, K.-C. (eds.) GPC 2007. LNCS, vol. 4459, pp. 519–530. Springer, Heidelberg (2007)
14. Gupta, B., Rahimi, S., Allam, V., Jupally, V.: Domino-Effect Free Crash Recovery for Concurrent Failures in Cluster Federation. In: Wu, S., Yang, L.T., Xu, T.L. (eds.) GPC 2008. LNCS, vol. 5036, pp. 4–17. Springer, Heidelberg (2008)
15. Qi, X., Parmer, G., West, R.: An Efficient End-Host Architecture for Cluster Communication. In: Proceedings of the 2004 IEEE Intl. Conf. on Cluster Computing, San Diego, California, pp. 83–92 (2004)
16. Shrivastava, S.K., Mancini, L.V., Randell, B.: The Duality of Fault- Tolerant System Structures. Software-Practice and Experience 23(7), 73–798 (1993)
17. Alvisi, L., Marzullo, K.: Message Logging: Pessimistic, Optimistic, and Causal. In: Proc. 15th IEEE Int. Conf. on Distributed Computing Systems, pp. 229–236 (1995)