Journal of Advance Computational Research
Vol. 3 No. 2 (July-December, 2018)

A High Performance Crash Recovery Approach for Distributed Systems

Bidyut Gupta
Department of Computer Science, Southern Illinois University, Carbondale, IL, USA

RECEIVED: December 29,2017. Revised March 06, 2018

Abstract: In this paper, we have proposed a new approach for checkpointing and recovery for concurrent failures in distributed computing environment. The proposed idea enables a process to restart from its recent checkpoint and hence ensures the least amount of re-computation after recovery. It also means that a process needs to save only its recent local checkpoint. The proposed value of the common check pointing interval enables an initiator process to log the minimum number of messages sent by each application process. The message complexity of the proposed check pointing algorithm as well as the recovery approach is *O*(n). Analytical performance-comparison with noted existing works also highlights the advantages of our scheme.

Keywords: Lost message; orphan message; checkpoint; concurrent failures; recovery

1. INTRODUCTION

Checkpointing and rollback recovery techniques are used to allow a distributed computing to progress in spite of a failure [1]-[8]. 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 C is defined as a consistent global checkpoint (state) if no message is sent after a checkpoint of C and received before another checkpoint of [3]. The checkpoints belonging to a consistent global checkpoint are called globally consistent checkpoints (GCCs).

The two fundamental approaches for check pointing and recovery are: asynchronous approach and the synchronous approach. In the asynchronous approach, taking checkpoints is very simple since processes take their checkpoints independently. After a system recovers from a failure, a procedure for rollback-recovery attempts to build a consistent global checkpoint. The procedure of finding a consistent global checkpoint is usually quite complex because of the possible presence of domino effect. Synchronous check pointing approach assumes that an initiator process which is different from the application processes invokes the check pointing algorithm periodically to determine a consistent global checkpoint. All processes coordinate through the exchange of control messages to determine a consistent global checkpoint. This makes the checkpointing approach very complex, where as the recovery approach is very simple because there is no need to determine a consistent global checkpoint after a system recovers from a failure unlike in asynchronous approach. 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 in order to ensure correct computation. In this context, it may be noted that a message is known as an orphan if its sending event is not recorded in a checkpoint of the sender, but its receiving event is recorded in a checkpoint of the receiver. A lost or delayed message is the one such that its sending event is recorded in a checkpoint of the sender, but because of a failure either the receiving event is not yet recorded in a checkpoint of the receiver, or the message does not arrive at the receiver. So to ensure correct computation of an application program, all these messages must have to be considered when a system restarts after a failure.

Problem formulation: In this work we address the following problem: given the respective recent local checkpoints of all 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 (recent) checkpoints which form together the recent consistent global checkpoint of the distributed system.

Note that to ensure correct computation all the above mentioned different types of messages have to be handled properly. Our work will be independent of the number of processes that may fail concurrently.

To fulfill our objective, we aim at designing a two-fold scheme stated as follows: first a single phase non-blocking check pointing approach will be considered that will ensure the non-existence of any orphan message with respect to the checkpoints of the application processes; second a recovery approach will be designed which will take care of any lost messages with respect to the recent checkpoints of the processes.

We shall use the following idea about the duration of the check pointing interval T: the time between two consecutive invocations of the check pointing algorithm, T is larger than the maximum message passing time between any two processes in the system. The impact of this idea will be clear when we discuss delayed and lost messages in Section 2.2. Note that the above idea about the check pointing interval was introduced to handle recovery in cluster computing environment [16].

2. RELEVANT DATA STRUCTURES AND CHECK POINTING INTERVAL

The distributed system has the following characteristics [1], [9], [10]: processes do not share memory and they communicate via messages sent through channels; processes are deterministic and fail stop.

2.1. Relevant Data Structures

Consider a set of n processes $\{P_1, P_2, \ldots\ldots, P_n\}$ involved in the execution of a distributed algorithm. We assume that application messages are piggybacked with unique sequence numbers, i.e. 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 check pointing 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 $C_x{}^i$; 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) $V^i_{x(\text{sent})} = [S^{i1}_x, S^{i2}_x, S^{i3}_x, \ldots, S^{in}_x]$, 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) $V^i_{x(\text{recv})} = [R^{i1}_x, R^{i2}_x, R^{i3}_x, \ldots, R^{in}_x]$, where R^{ij}_x represents the largest sequence number of all messages received by P_i from P_j in the check pointing interval $(C^i_x - C^i_{x-1})$. Also $R^{ii}_x = 0$.

2.2. Check Pointing Interval

We now state the reason for considering the value of the common check pointing interval T to be just larger than the maximum message passing time between any two processes of the system. In our explanation we will follow the same logic as was first given in [16]. 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 check pointing 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 check pointing 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 Fig. 1. 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 check pointing interval, any message m sent by process P_i during its check pointing 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 can not 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 process checkpoints of the processes and the messages logged at these recent checkpoints 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 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. Note that in the above discussion, we have implicitly assumed the non-existence of any abnormally excessive delay in message communication that violates our logical assumption that any message m sent by process P_i during its check pointing interval ($C^i_{x-1} - C^i_{x-2}$) always arrives before the recent checkpoint C^j_x of P_j.

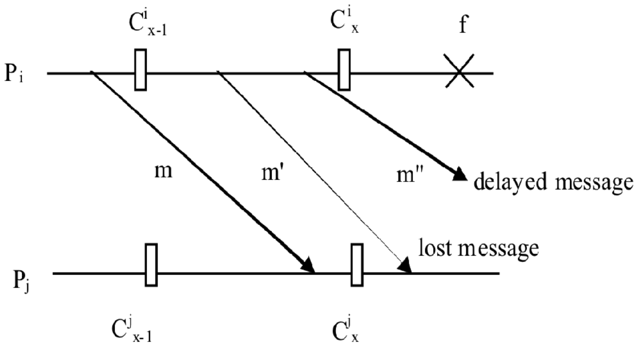

Figure 1: Message *m* Cannot be a Delayed Message

3. PROBLEMS ASSOCIATED WITH NON-BLOCKING APPROACH

It is known that the classical synchronous check pointing scheme 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; however it does have some problem. We explain first the problem associated with non-blocking synchronous check pointing 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 check pointing 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 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 check pointing 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. That is why we consider it as a single phase algorithm.

4. THE CHECK POINTING AND RECOVERY ALGORITHMS

Assume that it is the xth invocation of the check pointing algorithm. The algorithm produces *n* globally consistent checkpoints for a distributed system with n processes.

Algorithm Non-blocking

```
At each process $P_i$ ($1 \le i \le n$)
    if $P_i$ receives $M_c$
        takes checkpoint $C^i_x$;
        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 check pointing algorithm
            takes checkpoint $C^i_x$ without waiting for $M_c$;
            continues its normal operation;
            // processes the received message $m$ and ignores $M_c$, when received later
```

Proof of Correctness: In the 'if' block every process P_i takes its xth checkpoint C^i_x 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^i_x 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 can not be an orphan as well. Since this is true for all the processes, hence the recent xth checkpoints C^i_x, $1 \le i \le n$ are globally consistent checkpoints.

4.1. Performance of the Checkpoint Approach

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. About message complexity the initiator process broadcasts M_c only once. So the message complexity is $O(n)$.

Comparison with some noted existing works: We use some analytical results from [8] to compare our algorithm with some of the most notable algorithms in this area of research, namely [1], [7], and [8].

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 check pointing time;

Table 1
System Performance

Algorithm	*Blocking time*	*Messages*	*Distributed*
Koo-Toueg [1]	$n_{min} * T_{ch}$	$3 * n_{min} * n_{dep} * C_{air}$	Yes
Elnozahy [7]	0	$2 * C_{broad} + n * C_{air}$	No
Gao-Singhal [8]	0	$\approx 2 * n_{min} * C_{air} + \min(n_{min} * C_{air}, C_{broad})$	Yes
Our Algorithm	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 system. In Fig. 2, 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).

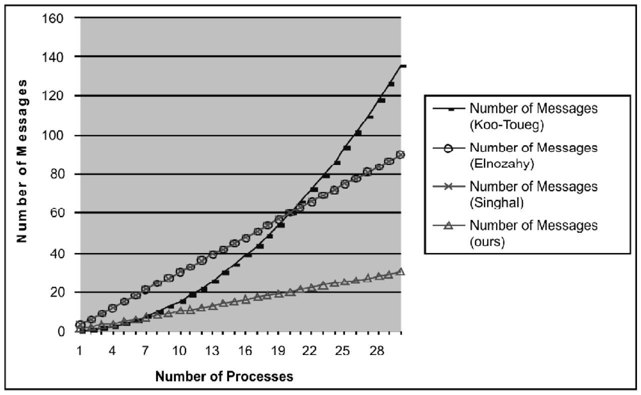

Figure 2: Number of Messages vs. Number of Processes for Four Different Approaches

4.2. 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 classical centralized approach [13] for message logging: all application messages are routed through the initiator process P_I. But, we differ from the centralized approach in that the messages sent to a process P_k are logged at P_I according to the order of their arrival at P_I and some of these messages may become lost messages in the event of a failure. This is a major difference because the approach in [13] only logs copies of the messages which have been exchanged between any two processes with the help of an extra acknowledgment protocol. In our work we denote this message log for process P_k as MESG_k, where $1 \leq k \leq n$ for an n process distributed system. Another major difference is that in our work the initiator process P_I 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 P_I. 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, each application process sends its sent and received vectors at its recent (say xth) checkpoint to P_I. Thus P_I gets all the n sent and n received vectors from the n application processes. Using these vectors P_I determines the lost messages, if any, sent by all other processes, P_i $(1 \leq i \leq n, i \neq k)$ to each P_k in the interval $(C^i_x - C^i_{x-1})$ in the following way:

Algorithm Recovery

```
For each process $P_k$ and $1 \leq i \leq n, i \neq k$
    if $S^{ik}_x > R^{ki}_x$
        $P_I$ records these sequence numbers $(R^{ki}_x + 1)$ to $S^{ik}_x$ in lost-from-$P^k_i$;
        // messages with sequence numbers $(R^{ki}_x + 1)$ to $S^{ik}_x$ are the lost messages from $P_i$ to $P_k$.
    $P_I$ forms the total order of all lost messages sent by every $P_i$, $i \neq k$ to $P_k$
    using lost-from-$P^k_i$ and the message log $\text{MESG}_k$ for $\text{P}_k$;
    $P_I$ sends to each $P_k$ the lost messages following their total order;
```

Theorem 1: Algorithm Non-blocking together with the recovery scheme results in correct computation of the underlying distributed application.

Proof: According to the check pointing 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 exixt any orphan or lost message with respect to the recent checkpoints. Hence the correctness of the underlying distributed computation is ensured.

4.3. Performance of the Recovery Approach

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. It also means that processes save only their recent checkpoints replacing their last 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 n processes, which carries the sent and received vectors of this process at its recent checkpoint. Therefore it needs only n control messages and so the message complexity is O(n). 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.

Comparison with Some Noted Existing Works

In the work [6] 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 can not be an orphan. In our work it is not necessary because of the check pointing scheme. Hence we avoid some unnecessary comparisons involved in such checking. The message overhead in [6] 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 [6] 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 [11] the following differences are observed. In [11] there is always an extra control messages 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 the work in [11] has the restriction that during normal computation receiver of a message can not send a new message until it receives the acknowledgement 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 failures in [11].

The work in [12] 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 check pointing 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 work in [15] introduced the idea of optimistic recovery. However, the recovery scheme proposed may suffer from the domino effect which may cause processes to roll back exponential number of times with respect to the number of processes. Also it considers only single failures. Our work is domino-effect free and also can handle concurrent failures. The work in [14] has presented an optimistic recovery algorithm that has a message complexity of $O(n^2)$ and a message overhead of $O(1)$. In our work we don't have any message overhead and also the message complexity of the proposed check pointing algorithm as well as the recovery approach is $O(n)$. Below in Table 2 we state a brief summery of comparisons of some important features of the the different check pointing / recovery approaches.

Table 2
Brief Summary of Comparisons

	Required Message ordering	*Maximum rollbacks Per failure*	*Message Overhead*	*Message Complexity*	*Number of concurrent Failures*
Manivannan[6]	None	1	$O(F)$	$O(n^2)$	n
Johnson[11]	None	1	$O(1)$	$O(n)$	1
Juang[14]	None	1	$O(1)$	$O(n^2)$	n
Damini[12]	None	1	$O(n)$	$O(n^2)$	n
Our Algorithm	None	1	None	$O(n)$	n

5. CONCLUSIONS

In this work, we have proposed a check pointing 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 most of the existing works. 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. It also means that processes can restart from their respective recent checkpoints. Our work is independent of the number of processes that may fail concurrently. Besides, the message complexity of the proposed check pointing algorithm as well as the recovery approach is just $O(n)$. Finally, note that our check pointing 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. Analytical performance-comparison with noted existing works highlights the advantages of our proposed schemes. Future work is directed at studying the effect of message sizes on the assumed check pointing intervals through simulation.

References

[1] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE trans. Software Engineering*, **SE-13**(1), 23-31, 1987.

[2] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Trans. Computing Systems*, **3**(1), 63-75, 1985.

[3] Y. Wang, “Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints”, *IEEE Trans. Computers*, **46**(4), 456-468, 1997.

[4] B. Gupta, S. K. Banerjee and B. Liu, “Design of New Roll-forward Recovery Approach for Distributed Systems”, *IEE Proc. Computers and Digital Techniques*, **149**(3), 105-112, 2002.

[5] B. Gupta, S. Rahimi, and Z. Liu, “A Novel Low-Overhead Roll-Forward Recovery Scheme for Distributed Systems”, *IET Computers and Digital Techniques*, **1**(4), 397-404, 2007.

[6] D. Manivannan and M. Singhal, “Asynchronous Recovery without using Vector Timestamps”, *Journal of Parallel and Distributed Computing*, **62** (2002), 1695 – 1728.

[7] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, “The Performance of Consistent Check pointing”, Proc. 11th Symp. on Reliable Distributed Systems, pp. 86-95, 1992.

[8] G. Cao and M. Singhal, “Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems”, *IEEE Transactions on Parallel and Distributed Systems*, **12**(2), 157–172, 2001.

[9] S. Venkatesan, T. Juang, and S. Alagar, “Optimistic Crash Recovery Without Changing Application Messages”, *IEEE Trans. Parallel and Distributed Systems*, **8**(3), 263-271, 1997.

[10] Pankaj Jalote, Fault Tolerance in Distributed Systems, PTR Prentice Hall, (1994), Addison-Wesley, (1998).

[11] D. B. Johnson and W. Zwaenepoel, “Sender-Based Message Logging”, Proc. 17th Fault-Tolerant Computing Symposium, Pittsburgh, pp. 14-19, 1987.

[12] O. P. Damini and V. K. Garg, “How to Recover Efficiently and Asynchronously When Optimism Fails”, Proc. 16th International Conference on Distributed Computing Systems, pp. 108-115, 1996.

[13] M. L. Powell and D. L. Presotto, “Publishing: A Reliable Broadcast Communication Mechanism”, Proc. 9th ACM Symposium on Operating Systems, 1983.

[14] T. Y. Juang and S. Venkatesan, “Efficient Algorithm for Crash Recovery in Distributed Systems”, Proc. 10th Conference on Foundations on Software Technology and Theoretical Computer Science, pp. 349-361, 1990.

[15] R. E. Strom and S. Yemini, “Optimistic Recovery in Distributed Systems”, *ACM transactions of Comput. Systems* **3**(3), 204-226, 1985.

[16] B. Gupta, S. Rahimi, V. Allam, and V. Jupally, “Domino-Effect Free Crash Recovery for Concurrent Failures in Cluster Federation”, Lecture Notes in Computer Science, **5036** (2008), 4-17.