Checkpointing and Rollback Recovery in Distributed Systems: Existing Solutions, Open Issues and Proposed Solutions

D. Manivannan
Department of Computer Science
University of Kentucky
Lexington, KY 40506

Abstract:- Checkpointing and rollback recovery are well-established techniques for dealing with failures in distributed systems. In this paper, we briefly summarize the existing solution approaches to these problems and also discuss the open issues, suggested approaches and some preliminary work that we have done addressing the open issues.

Keywords:- checkpointing, rollback recovery, consistent global checkpoint, failure recovery, fault-tolerance, zigzag paths.

1 Introduction

Distinction between distributed and parallel computing is diminishing. Our current day fastest parallel computers are distributed systems. In distributed and parallel computing systems, checkpointing and rollback recovery are well-known techniques for handling failures [1–8]. Existing checkpointing algorithms can be classified into three main categories – *asynchronous*, *synchronous* and *quasi-synchronous*. Under *asynchronous* checkpointing, processes take checkpoints periodically (i.e., the state of the processes are saved in stable storage periodically) without any coordination with other processes. When a process fails, a consistent global checkpoint is established and the processes are restarted from the most recent such consistent global checkpoint. Some advantages of this approach are: (i)it allows processes to take checkpoints whenever they want and hence processes could schedule their checkpointing activity when the I/O nodes are not busy and hence it causes less checkpointing overhead. However, it has the following disadvantages: (i) some or all the checkpoints taken may become useless. i.e., none of them may be part of any consistent global checkpoint; (ii) determining a consistent global checkpoint may involve lot of overhead [9], especially in large systems, and the processes may have to be restarted from the beginning due to the non-existence of a consistent global checkpoint other than the initial state (this is because in the worst case none of the checkpoints taken may be part of any consistent global checkpoint and hence useless).

In *synchronous or coordinated* checkpointing schemes, processes coordinate their checkpointing activity so that a globally consistent set of checkpoints is always maintained in the system. Due to its simplicity in implementation, synchronous checkpointing is widely used in supercomputers to cope with failures and it generally involves the following phases [10]: There is one process, called the coordinator, which coordinates the checkpointing activity. First, the coordinator process broadcasts a "quiesce" message to all processes involved in the computation. Upon receiving this message, each process stops its activities at a consistent and interruptible state and replies "ready" to the coordinator. After receiving the "ready" message from all processes, the coordinator broadcasts a "checkpoint" message to all the processes. Upon receiving the "checkpoint" message, each process dumps its state to an Input/Output (I/O) node and sends a "done" message to the coordinator. After the coordinator receives the "done" message from all the processes, the coordinator broadcasts a "proceed" message to all the processes. Upon receiving the "proceed" message, each process continues its activity from the point where it quiesced at which point the I/O nodes can write the checkpoint to the file system in the background.

Synchronous checkpointing has the following drawbacks, especially when the number of processes involved is very large:

Message overhead: This approach will be efficient if the number of processes involved in the computation is small, say in 100s. If the number of processes involved in the computation is in 100s of thousands, then this approach will cause lot of message overhead.

Checkpointing Overhead: In general, the number of I/O nodes are much less when compared to the number of processes in the system. After a process sends a "ready" message, it quiesces its activities until it receives "proceed" message from the coordinator, during which time no useful work is done by the processes. Moreover, when the checkpoint of a process is dumped to an I/O node, it can cause lot of contention for the I/O nodes since one I/O node supports several processors. All this contribute to checkpointing overhead (i.e., the time spent by the processes without doing any useful work).

Under *quasi-synchronous* (or *communication-induced*) checkpointing, processes are allowed to take checkpoints (called *basic checkpoints*) independently in their assigned time

slots or whenever a local predicate becomes true, as well as reduce the number of useless checkpoints by taking additional checkpoints (called *forced checkpoints*) at appropriate times. Hence, communication-induced checkpointing algorithms allows the processes to take checkpoints whenever they want while at the same time all the checkpoints taken are made useful and hence have the advantages of both synchronous and asynchronous checkpointing algorithms.

Paper Objectives and Organization

In this paper, we briefly revisit the approaches taken for checkpointing and rollback recovery in distributed systems. We first briefly review our classification of quasi-synchronous checkpointing algorithms and its impact in this area. Then we address some of the open issues that exist in this area and our preliminary work that addresses these open issues. The rest of the paper is organized as follows. In Section 2, we present the preliminary background required for the paper. In Section 3, we briefly review our characterization and classification of quasi-synchronous checkpointing algorithms and its impact. In Section 4, we discuss the open issues and proposed approaches for addressing these open issues. In Section 5, we present our preliminary work in this direction. Section 6 concludes the paper.

2 Preliminaries

2.1 System Model

The distributed computation we consider consists of N spatially separated sequential processes denoted by P_1, P_2, $\cdots$, P_N. The processes do not share a common memory or a common clock. Message passing is the only way for processes to communicate with one another. The computation is asynchronous: each process progresses at its own speed and messages are exchanged through reliable communication channels, whose transmission delays are finite but arbitrary.

Execution of a process is modeled by three types of events – the send event of a message, the receive event of a message and an internal event. The states of processes depend on one another due to interprocess communication. Lamport's *happened before* relation [11] on events, $\xrightarrow{hb}$, is defined as the transitive closure of the union of two other relations: $\xrightarrow{hb} = (\xrightarrow{xo} \cup \xrightarrow{m})^+$. The $\xrightarrow{xo}$ relation captures the order in which local events of a process are executed. The i^{th} event of any process P_p (denoted $e_{p,i}$) always executes before the $(i+1)^{st}$ event: $e_{p,i} \xrightarrow{xo} e_{p,i+1}$. The $\xrightarrow{m}$ relation shows the relation between the send and receive events of the same message: if a is the send event of a message and b is the corresponding receive event of the same message, then $a \xrightarrow{m} b$.

Each checkpoint taken by a process is assigned a unique sequence number. The $i^{th}(i \geq 0)$ checkpoint of process P_p is assigned the sequence number i and is denoted by $C_{p,i}$. We assume that each process takes an initial checkpoint before execution begins and a *virtual* checkpoint after execution ends. Sometimes, checkpoints are also denoted by the letters A, B, or C for clarity. The i^{th} checkpoint interval of process P_p is all the computation performed between its $(i-1)^{th}$ and i^{th} checkpoints (and includes the $(i-1)^{th}$ checkpoint but not i^{th}).

The send and the receive events of a message M are denoted respectively by $send(M)$ and $receive(M)$. So, $send(M) \xrightarrow{hb} C_{p,i}$ if message M was sent by process P_p before taking the checkpoint $C_{p,i}$. Also, $receive(M) \xrightarrow{hb} C_{p,i}$ if message M was received and processed by P_p before taking the checkpoint $C_{p,i}$. $send(M) \xrightarrow{hb} receive(M)$ for any message M. Next, we present the definition of a consistent global checkpoint formally.

Definition 1 *A set $S = \{C_{1,m_1}, C_{2,m_2}, \cdots, C_{N,m_N}\}$ of N checkpoints, one from each process, is said to be a* **consistent global checkpoint**[1] *if for any message M and for any integer p, $1 \leq p \leq N$: $receive(M) \xrightarrow{hb} C_{p,m_p} \Longrightarrow send(M) \xrightarrow{hb} C_{q,m_q}$ for some q, $1 \leq q \leq N$.*

2.2 Z-paths and their Properties

Netzer and Xu [12] gave a necessary and sufficient condition for a given set of checkpoints to be part of a consistent global checkpoint. They introduced the notion of *zigzag path*, which is a generalization of a *causal path*[2] induced by the Lamport's happened before relation. A zigzag path (or a *Z-path* for short) between two checkpoints is like a causal path, but a Z-path allows a message to be sent before the previous one in the path is received. Formally, a *Z-path* between two checkpoints is defined [9, 12] as:

Definition 2 *A Z-path exists from $C_{p,i}$ to $C_{q,j}$ if*

1. *$p = q$ and $i < j$ (i.e., the two checkpoints are from the same process and the former precedes the later) or*
2. *there exist messages $m_1, m_2, \cdots, m_n (n \geq 1)$ such that*
   - *(a) m_1 is sent by process P_p after $C_{p,i}$,*
   - *(b) if m_k $(1 \leq k < n)$ is received by P_r, then m_{k+1} is sent by P_r in the same or later checkpoint interval (although m_{k+1} may be sent before or after m_k is received), and*
   - *(c) m_n is received by P_q before $C_{q,j}$.*

An important property of Z-paths is that it captures the precise requirement for a set of checkpoints to be a part of a consistent global checkpoint as stated in the following theorem due to Netzer and Xu [12].

Theorem 1 *A set of checkpoints S can be extended to a consistent global checkpoint if and only if there is no Z-path between any two (not necessarily distinct) checkpoints in S.*

[1] Also called a **consistent global snapshot** or **a consistent cut**.

[2] A causal path from a checkpoint A to checkpoint B exists if and only if there exists a sequence of messages $m_1, m_2, \cdots, m_n$ such that m_1 is sent after A, m_n is received before B, and m_i is received by some process before the same process sends m_{i+1} $(1 \leq i < n)$.

Proof: Proof can be found in [12]. □

Theorem 1, presents the precise condition a set of checkpoints needs to satisfy in order to be part of a consistent global checkpoint. The following corollary of Theorem 1 gives the precise condition for a single checkpoint to be part of a consistent global checkpoint.

Corollary 1 *A checkpoint C can be part of a consistent global checkpoint if and only if it is not on a Z-cycle.*

Proof: Follows from Theorem 1 by taking $S = \{C\}$. □

In quasi-synchronous checkpointing, processes take communication-induced checkpoints to reduce the number of useless checkpoints; the message pattern and knowledge gained about the dependency between checkpoints of processes trigger communication-induced checkpoints so that the number of useless checkpoints is minimized or eliminated. Let us first understand how checkpoints become useless and how we can convert useless checkpoints into useful checkpoints.

Definition 3 *A* **non-causal Z-path** *from a checkpoint $C_{p,i}$ to a checkpoint $C_{q,j}$ is a sequence of messages $m_1, m_2, \cdots, m_n (n \geq 2)$ satisfying the conditions of Definition 2 such that for at least one i $(1 \leq i < n)$, m_i is received by some process P_r after sending the message m_{i+1} in the same checkpoint interval.*

Thus, non-causal Z-paths are those Z-paths that are not causal paths; in particular, Z-cycles are non-causal Z-paths. Moreover, it is not possible to track non-causal Z-paths between checkpoints on-line, and hence the presence of non-causal Z-paths complicates the task of finding consistent global checkpoints. However, non-causal Z-paths between checkpoints are preventable if processes take additional checkpoints at appropriate times. Preventing all the non-causal Z-paths between checkpoints by making processes take additional checkpoints at appropriate times not only makes all the checkpoints useful but also facilitates construction of consistent global checkpoints incrementally and easily; this is because, in the absence of non-causal Z-paths, any set of checkpoints that are not pairwise causally related can be extended to a consistent global checkpoint by Theorem 1 and causality between checkpoints can be tracked on-line by using vector timestamps [13] or similar mechanisms.

Depending on the degree to which the non-causal Z-paths are prevented, quasi-synchronous checkpointing algorithms exhibit different properties and can be classified into various classes. This classification helps to understand the properties and limitations of various checkpointing algorithms which is helpful for comparing their performance; it also helps in designing more efficient algorithms. Next, we review our classification of quasi-synchronous checkpointing algorithms.

3 Classification of Quasi-Synchronous Checkpointing

We first review our classification [2] of quasi-synchronous checkpointing algorithms. This classification is based on the degree to which the formation of non-causal Z-paths are prevented.

Strictly Z-path Free Checkpointing

Definition 4 *A checkpointing pattern is said to be strictly Z-path free (or SZPF) if there exists no non-causal Z-path between any two (not necessarily distinct) checkpoints.*

In a SZPF system, since non-causal Z-path are eliminated, a message sequence forms a Z-path if and only if it forms a causal path. This means that in every checkpoint interval, all the message-receive events precede all the message-send events. So, if checkpoints are forced in such a way that no message-send even occurs before a message-receive event in the same checkpoint interval, then the system will be SZPF. Since a SZPF system does not allow non-causal Z-paths, a set of checkpoints S can be extended to a consistent global checkpoint if and ony if no two checkpoints in S are causally related. Thus, since causality can be tracked on-line using vector timestamps or similar other mechanisms [13], we can incrementally extend any set of checkpoints S that are not pairwise causally related to a consistent global checkpoint.

Z-path Free Checkpointing

A ZPF system, allows non-causal Z-paths as long as there are causal siblings for such paths. More precisely, A checkpointing pattern is said to be Z-path free (or ZPF) if and ony if for any two checkpoints A and B, there exists a Z-path from A to B if and ony if $A \xrightarrow{\text{hb}} B$. A checkpointing algorithm that makes the systems ZPF has potential to induce less checkpointing overhead while at the same time maintain all the desirable properties of a SZPF system. The ZPF property is equivalent to Rollback Dependency Trackablity, proposed by Wang [14].

Z-cycle Free Checkpointing

All checkpoints taken in a ZPF system and a SZPF system are useful. If the objective of a quasi-synchronous checkpointing algorithm is just to make all checkpoints useful, it is not necessary to make the system either ZPF or SZPF. To make all checkpoints useful, it is sufficient to prevent only Z-cycles 1. In otherwords, a checkpointing pattern is said to be Z-cycle free (or ZCF) if and only if none of the checkpoints lies on a Z-cycle.

An important feature of a ZCF system is that every checkpoint taken is useful, since none of the checkpoints is on a Z-cycle. Even though every checkpoint in a ZCF system is useful, constructing a consistent global checkpoint incrementally is difficult due to the presence of non-causal Z-paths, which

are difficult to track on-line. There is a trade off between weaker checkpointing model and the easeness of constructing consistent global checkpoints.

Z-cycles cannot be tracked on-line [2,15]. However, if all Z-cycles are prevented, then all the checkpoints taken are useful. This prompted many researchers to design communication-induced checkpointing algorithms that prevent Z-cycles [16–18]. The Checkpointing protocol proposed by Helary et al. [8, 19] tries to exploit as much information as possible from the causal past and make informed decision to take as less forced checkpoints as possible. This is why this protocols is often called in the literature [20,21] as "Fully informed communication-induced checkpointing protocol". However, since designing an optimal communication-induced checkpointing algorithm that prevents all Z-cycles is not possible (because Z-cycles cannot be tracked on-line), all these algorithm try to force processes to take additional checkpoints whenever there is a potential (according to the heuristics designed by the individual researchers) for the formation of Z-cycles. Such algorithms were generally evaluated through simulation. A communication-induced checkpointing algorithm that incurs lower forced checkpoints is considered to be better.

4 Open Issues

Quasi-synchronous checkpointing mitigates some of the problems with synchronous and asynchronous checkpointing methods such as making every checkpoint useful and reducing the message overhead. However, contention for stable storage is still a problem when several processes take checkpoints simultaneously and hence the existing communication-induced checkpointing algorithms do not scale well. This will have a great impact on the checkpointing *overhead* and extend the total execution time of the distributed computation significantly in large scale distributed systems.

Thus, we need to revisit this problem from the perspective of future generation large scale parallel/distributed systems. One approach proposed in the literature for handling the contention problem in the case of synchronous checkpointing is to stagger checkpointing [22]. *Checkpoint staggering* attempts to prevent two or more processes taking checkpoints at the same time and reduce contention for accessing I/O nodes for storing checkpoints. To the best of our knowledge, checkpoint *staggering* has previously been proposed for only *coordinated* checkpointing algorithms [22]. However, if all the processes quiesce until all the processes take checkpoints, checkpoint staggering is not suitable for large scale distributed systems containing hundreds of thousands of nodes. Next, we briefly describe some possible approaches for addressing these issues and also present some preliminary work we have done in this area.

5 Possible Approaches

Following are some possible approaches for designing scalable checkpointing and recovery algorithms for future generation supercomputers.

Synchronous Checkpoint Staggering: Under this approach, a process involved in the distributed computation initiates checkpointing after saving its state. Other processes save their state in such a way that saving their state does not cause contention for the I/O nodes. Note that in a given system that has thousands of nodes only a few nodes will serve as I/O nodes(For example, in Blue-Gene/L [23], the number of computing nodes per I/O node is 64, average bandwidth between a I/O nodes and compute nodes is 350 MBps, filesystem bandwidth for I/O node is 1GBs and average checkpoint size of a typical process involved is 256MB). In this approach, processes need to be assigned timeslots to take basic checkpoints. Since there is no global clock, due to clock skew, even if two processes are assigned two different timeslots they could still end up taking checkpoints at the same time causing contention. Moreover, since processes are not quiesced during checkpointing, due to dependencies created by messages exchanged by processes some processes may be forced to take checkpoint before their scheduled time which can cause contention and hence increase checkpointing overhead.

Communication-Induced Checkpoint Staggering: If we can schedule the basic checkpoints in such a way that it prevents contention for stable storage, then such algorithms can be made scalable. However, forced checkpoints taken, depending on the communication pattern of the application, may cause contention for stable storage hindering such algorithms from scaling.

Memory-based Checkpointing: As we mentioned earlier, since storing the checkpoints of processes in files can cause contention for the I/O nodes and increase the checkpointing overhead, it is worthwhile to explore the possibility of designing checkpointing algorithms which allow checkpoints of a process to be stored in a neighboring node's memory. Thus, when a node fails, checkpoints of processes that were running on the failed node can be retrieved from the memory of the neighboring nodes, assuming that the probability of the neighboring nodes failing at the same time is small. Concurrent failure of neighboring can be handled if we can use methods such as the ones used in RAID (Redundant Array of Inexpensive Disks) by storing checkpoints in the memory of multiple neighboring nodes using the RAID technique.

Clustering Approach: Many distributed applications exhibit a communication model which is cluster-based. In other words, each process involved in the computation communicates only with the nodes in its cluster most of the time. For such applications, designing synchronous checkpointing at the cluster level and using a communication-induced approach at the entire system level could decrease the checkpointing overhead.

Next we briefly discuss some preliminary work we have done.

5.1 One Approach

First, we present a staggered quasi-synchronous checkpointing algorithm which not only makes all checkpoints useful but also reduces contention for stable storage by taking basic checkpoints (checkpoints taken by a process voluntarily) in a staggered manner [5]. Since all checkpoints taken are useful, the algorithm ensures the existence of a *recovery line* (i.e., a consistent global checkpoint) containing any checkpoint of any process. This property of the algorithm helps bound rollback during recovery due to a failure. We only informally describe the algorithm.

5.2 Informal Description of the Algorithm

Under our algorithm, each process takes basic checkpoints independently in their assigned time slots. In addition, to prevent useless checkpoints, processes take forced checkpoints upon the reception of some messages. Each checkpoint is assigned a unique sequence number. The sequence number assigned to a basic checkpoint is the current value of a local counter (an integer variable). Since the sequence numbers assigned to basic checkpoints are picked from the local counters which are incremented periodically, the sequence numbers of the latest checkpoints of all the processes will differ by at most one as long as the local clocks do not drift more than half the checkpoint time interval. This property helps in advancing the recovery line. When a process P_p sends a message, it appends the sequence number of its current checkpoint to the message. When a process P_q receives a message, if the sequence number appended to the message is greater than the sequence number of the latest checkpoint of P_q, then, before processing the message, P_q takes a checkpoint and assigns the sequence number received in the message as the sequence number of the checkpoint taken. When it is time for a process to take a basic checkpoint, it skips taking a basic checkpoint if its latest checkpoint has a sequence number greater than or equal to the current value of its counter (this situation could arise as a result of the forced checkpoints or drift in local clocks). This strategy helps to reduce the the total number of checkpoints taken. An alternative approach to reduce the number of checkpoints would be to allow a process to delay processing a received message until the sequence number of its latest checkpoint is greater than or equal to the sequence number received in the message, if the application allows that.

If several processes take checkpoints simultaneously, they will contend for access to the stable network storage. Contention can be reduced by allowing processes to take checkpoints in a staggered manner. Next, we illustrate our approach for taking basic checkpoints in a staggered manner. We assume that there are a total of N processes $P_0, P_1, \ldots, P_{N-1}$, involved in the distributed computations we consider. Each process has a unique process id. For example, process P_p (where $0 \leq p < N$) has process id p. We also assume that it takes at most t (maximum checkpoint latency) time units to take a checkpoint and send it to the stable network storage in the absence of contention for stable storage. Each process takes one checkpoint (either basic or forced) within each checkpoint interval X. A local variable $next_p$, which is incremented by 1 at the end of each checkpoint interval, keeps track of the current number of checkpoint intervals. We denote the local clock at the site in which process P_p is running as C_p. The current time at clock C_p is denoted by $V(C_p)$. For simplicity, we assume that $V(C_p)$ is initialized to 0.

Within each checkpoint interval of length X time units, a process takes a basic checkpoint some time during the second half of the interval if it has not taken a forced checkpoint in that interval. The second half of the interval is divided into several time slots. The size of each time slot T is at least t (maximum checkpoint latency) plus δ (maximum local clock drift) time units. So, T is defined as follows:

$$T = t + \delta \tag{1}$$

The number of slots within a checkpoint interval, denoted by γ, is given by Equation 2.

$$\gamma = \lfloor X/(2T) \rfloor \tag{2}$$

We assume that X is chosen such that T is much smaller than X (i.e., $T << X$). For example, if $N \leq 15$ and $T = 10$ seconds, it may be ideal to choose $X = 5$ minutes so that there would be 15 available slots during the later half of each 5 minute checkpoint interval and each process can have its own time slot to take a basic checkpoint. However, if there are more processes than time slots, then we use a round-robin method to reduce contention for stable storage. This can be achieved as follows:

> A process P_p takes a basic checkpoint when its local time $V(C_p) = (next_p - 1) * X + X/2 + (p \bmod \gamma) * T$ if there is no forced checkpoint already taken in the period of time from $(next_p - 1) * X$ to $V(C_p)$.

So far we discussed how our algorithm takes basic checkpoints in a staggered manner to reduce contention. It is also possible to reduce contention between basic and forced checkpoints. Due to space limitation, we do not discuss these optimizations or results of our performance evaluation.

6 Conclusion

In this paper, we surveyed briefly the classification of checkpointing algorithms and also addressed some open issues that remain to be addressed. Then, we presented some of our preliminary work done in this area.

Acknowledgement

This material is based in part upon work supported by the US National science Foundation under Grant No. IIS-0414791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Manivannan and M. Singhal, "A Low-overhead Recovery Technique Using Quasi-synchronous Checkpointing," in *Proceedings of the* 16^{th} *IEEE International Conference on Distributed Computing Systems*, Hong Kong, May 1996, pp. 100–107.

[2] ——, "Quasi-Synchronous Checkpointing: Models, Characterization, and Classification ," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 703–713, July 1999.

[3] ——, "Asynchronous Recovery Without Using Vector Timestamps," *Journal of Parallel and Distributed Computing*, vol. 62, no. 12, pp. 1695–1728, December 2002.

[4] Q. Jiang and D. Manivannan, "An Optimistic Checkpointing and Message Logging Approach for Consistent Global Checkpoint Collection in Distributed Systems," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, Long Beach, California, March 2007.

[5] J. Y. D. Manivannan, Q. Jiang and M. Singhal, "A Quasi-Synchronous Checkpointing Algorithm that Prevents Contention for Stable Storage," *Information Sciences*, 2008.

[6] A.Agbaria, H.Attiya, R. Friedman, and R. Vitenberg, "Quantifying Rollback Propagation in Distributed Checkpointing ," *Journal of Parallel and Distributed Computing*, vol. 64, pp. 370–384, 2004.

[7] E. Elnozahy and J. Plank, "Checkpointing for Peta-scale Systems: A Look into the Future of Practical Rollback-Recovery ," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 97–108, 2004.

[8] J. Tsai, S. Kuo, and Y. Wang, "More Properties of Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability," *Journal of Information Science and Engineering*, vol. 21, pp. 239–257, 2005.

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

[10] L. Wang, K. Pattabiraman, Z. Kalbarczyk, ravishankar K Iyer, L. Votta, C. Vick, and A. Wood, "Modeling Coordinated Checkpointing for Large-Scale Supercomputers," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05).*, 2005.

[11] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems," *Communications of the ACM.*, vol. 21, no. 7, pp. 558–565, July 1978.

[12] R. H. B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 165–169, February 1995.

[13] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms*, M. C. et al., Ed. Elsevier Science, North Holland, 1989, pp. 215–226.

[14] 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, April 1997.

[15] J. Tsai, Y.-M. Wang, and S.-Y. Kuo, "Evaluation of Domino-Free Communication-Induced Checkpointing Protocols," Microsoft Corporation, http://www.research.microsoft.com/scripts/pubdb/trpub.asp, Res. Rep MSR-TR-98-43, September 1998.

[16] R. Baldoni, F. Quaglia, and B. Ciciani, "A VP-Accordant Checkpointing Protocol Preventing Useless Checkpoints," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed System*, October 1998.

[17] J.-M. Hélary, A. Mostéfaoui, and M. Raynal, "Communication-induced determination of consistent snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 9, pp. 865–877, 1999.

[18] I. C. Garcia and L. E. Buzato, "On the minimal characterization of the rollback-dependency trackability property," *21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, vol. 00, p. 0342, 2001.

[19] J.-M. Hlary, A. Mostefaoui, and M. Raynal, "Communication-Induced Determination of Consistent Snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 9, pp. 865–877, 1999.

[20] J.C.Tsai, "An efficient Index-based Checkpointing Protocol with Constant-size Control Information on Messages ," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, pp. 287–296, 2005.

[21] J. Tsai and J.-W. Lin, "On the fully-informed communication-induced checkpointing protocol," *PRDC '05: 11th Pacific Rim International Symposium on Dependable Computing*, pp. 151–158, 2005.

[22] N. Vaidya, "Staggered Consistent Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 694–702, January 1999.

[23] N.R.Adiga et al., "An Overview of the Blue Gene/L," in *Proceedings of the IEEE International Conference on Supercomputing*, 2002.