Hierarchical Coordinated Checkpointing Protocol

Himadri S. Paul * Arobinda Gupta R. Badrinath
Department of Computer Science and Engineering,
Indian Institute of Technology,
Kharagpur, INDIA - 721302.
email:`<hpaul,agupta,badri>@cse.iitkgp.ernet.in`

ABSTRACT
Coordinated checkpointing protocol is a simple and useful protocol, used for fault tolerance in distributed system on LAN. However, checkpoint overhead of the protocol is bottlenecked by the link speed. Checkpoint overhead of the protocol increases even if only one link in the network is of low-speed. In a metacomputing environment, where distributed application communicates over low speed WAN, the checkpoint overhead becomes very large. In this paper we present *hierarchical coordinated checkpointing protocol* which aims to overcome the network speed bottleneck. The protocol is based on the *2-phase commit protocol*. The protocol is suitable for an internet-like network topology, where clusters of computers are connected via high speed link and the clusters are connected through low-speed links. Metacomputing environment runs over similar networks. We present simulation studies of the protocol, and it shows checkpoint overhead improvement over that of the well-known coordinated checkpointing protocol.

KEY WORDS
Checkpoint and Rollback Recovery, Fault-tolerant Systems, Distributed Algorithm.

1 Introduction

Fault tolerance can be used to save computation wastes for long running distributed applications. Fault tolerance through checkpoint is a convenient method used in many distributed systems, like distributed shared memory systems [8], Fail-safe PVM [7], MIST [1], Globus [5], the Legion system [4], etc.

Coordinated checkpointing protocol is a simple protocol used in many distributed systems for fault tolerance. However, the checkpoint overhead of coordinated protocol increases with link latency [9]. Performance of checkpointing protocol in systems distributed over a wide-area like metacomputers, is bottlenecked by the low-speed links in the network. In this paper we propose a checkpoint and recovery protocol for such distributed systems, which is based on the well-known *2-phase commit protocol*. The proposed protocol, named *hierarchical coordinated checkpointing protocol*, tries to minimize checkpointing overhead in distributed systems running on networks like the internet. We show, by simulation, that the proposed protocol achieves significant reduction in checkpoint overhead over the conventional coordinated checkpoint protocol.

In the next section we discuss our motivation in designing the *hierarchical coordinated checkpointing protocol* in more details. In Section 3 we present the protocol, and provide a sketch of proof of correctness in Section 4. We have simulated both the coordinated checkpointing protocol and the hierarchical coordinated checkpointing protocol. We present comparative performance results in Section 5. Finally, Section 6 concludes the paper.

2 Motivation and Contribution

A *global checkpoint* of a distributed application consists of a set of local checkpoints, one contributed by each of the processes in the system and the state of the communication medium or channel. However, all global checkpoints are not consistent. A *consistent global checkpoint* or a *recovery line* is a global checkpoint, where a message shown as received by a local checkpoint must be shown sent by some other local checkpoint in the set [2]. The objective of a checkpoint and recovery protocol is to rollback the distributed system to a consistent global checkpoint at the event of a fault.

Several checkpoint and recovery protocol for distributed systems are available in literature [3]. Coordinated checkpointing is one of the early checkpointing methods proposed for distributed systems. This class of protocol ensures that whenever a process takes a local checkpoint, all other processes in the system also take their respective local checkpoints. Koo and Toueg discuss one such protocol in [6]. Coordinated protocol is popularly used for LAN-based distributed applications for fault tolerance. However, checkpoint overhead of coordinated protocol increases with link latency [9]. In a networked system, for example a metacomputing system, where link speeds are different in different parts of the network, the coordinated checkpointing scheme is bottlenecked by the low speed links.

The main emphasis in development of newer techniques of checkpoint is to minimize checkpoint and recovery overheads. To address the issue, we propose a *hierarchical coordinated checkpointing*, based on the coordi-

*Supported by a graduate fellowship from Infosys Technologies Ltd., Bangalore, INDIA.

nated checkpointing protocol (also referred in this paper as flat coordinated checkpointing). The protocol tries to reduce checkpoint overhead on clustered network topology. In the proposed protocol, we try to avoid explicit coordination throughout the system during checkpointing activity, and delegate the job to one representative process, called *leader*, in each of the clusters. It is then the responsibility of the cluster 'leader' to coordinate with the processes inside its cluster and establish checkpoint. The approach reduces message transmission over the low speed links, and reduces checkpointing overhead. We have carried out extensive simulation studies of both the hierarchical and flat coordinated protocol under different conditions. The results show considerable improvement in checkpoint overhead in case of the hierarchical coordinated protocol. The results will be discussed in Section 5. In the following section, we discuss the hierarchical checkpointing protocol based on the 2-phase commit protocol.

3 Hierarchical Coordinated Checkpointing Protocol

The Hierarchical checkpointing protocol is designed for hierarchical network topology, like the internet. We define a *cluster* as a collection of computing nodes ('node' and 'process' have been used interchangeable throughout the paper) connected via high speed links. The network consists of several clusters, and computers belonging to two different clusters are connected by slow speed link. All nodes are reachable from every other nodes. The nodes are assumed to be *fail-safe* [10]. All the links are unreliable. However we assume the network is immune to partitioning, *i.e.*, all the nodes in the network are eventually reachable from other nodes. The checkpoint and recovery layer resides in between the communication layer and the application layer. The application may not be aware of the cluster identity of all the nodes in the system, but all the lower layers do.

We assume that there is only one checkpoint initiator. For simplicity in presenting the protocol, we also assume that only one fault occurs at a time and no fault occurs during recovery process. However, these constraints can be easily overcome, and the required modifications are discussed at the end of Section 4. During the execution of the protocol a process assumes one of the following 3 roles during a checkpointing session.

- *Initiator:* the process which initiates a checkpointing session.
- *Leader:* one particular process from each cluster which is responsible for coordinating activities within the boundary of its own cluster, in accordance with the instructions from the initiator. The identity of the leaders are known to the initiator.
- *Follower:* the rest of the processes in the system. They follow instructions from the leader of its cluster.

Similarly, during recovery the processes assume one of the above roles. In this case, the failed process takes the role of *initiator*.

Each process stores one *permanent* checkpoint. In addition each process can also have one *tentative* checkpoint. However, the tentative checkpoints are transient, and are either discarded or made permanent after some time. Processes maintain a checkpoint sequence number (csn), and it is incremented by one in every checkpoint session. The csn is piggybacked on every control messages. In addition each process also maintains a boolean variable *in_ckpnt*, which indicates whether the process is in the checkpoint session or not. Also each process maintains another variable *epoch* to indicate the number of times it has undergone recovery process. The *epoch* is stored as part of checkpoint of the process. This value is also sent along with all control messages.

Blocking a process implies that the process is preempted into the 'wait' state, *i.e.* it does not execute. The process remains in the wait state until it is explicitly unblocked. We define a policy, B, which when invoked on a process blocks its the execution at its first attempt to send an application message to any extra-cluster process. Since all the messages pass through the checkpoint layer, it can enforce such a policy. Note the difference between 'blocking' a process and 'invoking policy B'. 'Blocking a process' immediately stops its execution. However to 'invoke policy B on a process' lets the process execute until it tries to send an extra-cluster application message.

The checkpointing protocol is divided into two parts, and both these parts execute in parallel. The first part of the protocol is a coordinated checkpointing protocol which operates within the boundary of the clusters. The cluster leaders in this part assume the role of checkpoint coordinator. This part of the protocol follows the well-known *2-phase commit protocol*. During this part of the protocol, the processes in a cluster remain blocked, and they establish a consistent checkpoint within the boundary of the cluster. The second part of the protocol is also a coordinated checkpointing protocol, but the cluster leaders are the only participants in this part of the protocol, with the initiator working as the coordinator. This part of the protocol runs in background. Until the termination of the second part, a cluster-wide blocking is implemented by virtue of policy B, and the processes are restricted from communicating with processes outside its own cluster. However, processes continue with their normal execution as long as they do not need to send a message outside their own cluster. The initiator initiates coordination for both part of the protocol in parallel. In this protocol the initiator process takes up the role of leader for its own cluster. A proper assignment of processes of the distributed system, such that more communicating processes in one cluster, will produce better checkpoint performance in this scheme. Below we formally present the hierarchical checkpointing protocol.

3.1 ALGORITHM: Checkpointing

Local Variables:

```
        int epoch ← 0.
        int csn ← 0.
        bool in_ckpnt ← FALSE.
```

Initiator

```
  role ← INITIATOR.
  $P \leftarrow \{p : p$ is an intra-cluster process$\}$.
  $L \leftarrow \{p : p$ is leader process of another cluster$\}$.
  $leader$ ← my process id.

C1:
  block application process.
  csn ← csn+1.
  in_ckpnt ← TRUE.
  send ckpt_req to all processes in $(P \cup L)$.
  wait till receive of ack_ckpt_req from all
   processes in $P$.
  send ckpt_estb to all processes in $P$.
  take a tentative checkpoint.
  wait till receive of ack_ckpt_estb from all
  process in $P$.
      Unblock application process.
      Invoke policy $B$ on application process.
  wait till receive ack_ckpt_req msgs from
    all process in $L$.
C2:
  revoke policy $B$.
  send commit_req to all process in $(P \cup L)$.
  make the tentative checkpoint permanent.
  wait till received ack_commit_req from all
    processes in $P \cup L$.
  in_ckpnt ← FALSE.
```

Leader:
(on receipt of *ckpt_req* from the initiator)

```
  role ← LEADER.
  $P \leftarrow \{p : p$ is an intra-cluster process$\}$.
  $leader$ ← initiator process.

C1:
  block the application process.
  csn ← csn + 1.
  in_ckpnt ← TRUE.
  send ckpt_req to all processes in $P$.
  wait till received ack_ckpt_req from
    all process in $P$.
  send ckpt_estb to all processes in $P$.
  take a tentative checkpoint.
  wait till received ack_ckpt_estb from
    all processes in $P$.
  unblock application processes.
  invoke policy $B$ on application process.
  send ack_ckpt_req to $leader$.
C2:
  wait for receipt of commit_req to $leader$.
  revoke policy $B$.
  send commit_req to all processes in $P$.
  make the tentative checkpoint permanent.
  wait till received ack_commit_req from
    all processes in $P$.
  send ack_commit_req to $initiator$.
  in_ckpnt ← FALSE.
```

Follower:
(on receipt of *ckpt_req* from
 the leader of the cluster)

```
  role ← FOLLOWER.
  $leader$ ← leader of the cluster.

C1:
  block application process.
  csn ← csn+1.
  in_ckpnt ← TRUE.
  send ack_ckpt_req to $leader$.
  wait for receipt of ckpt_estb from $leader$.
  take tentative checkpoint.
  invoke policy $B$ on application process.
  send ack_ckpt_estb to $leader$.
  unblock application processes.
C2:
  wait till receipt of commit_req from $leader$.
  revoke policy $B$.
  make the tentative checkpoint permanent.
  send ack_commit_req to $leader$.
  in_ckpnt ← FALSE.
```

3.2 ALGORITHM: Recovery

Initiator:
(failed process)

```
  role ← INITIATOR.
  $P \leftarrow \{p : p$ is an intra-cluster process$\}$.
  $L \leftarrow \{p : p$ is leader process of another cluster$\}$.
  in_ckpnt ← FALSE.
  if (tentative checkpoint present)
      make it a permanent checkpoint.
  endif.
  rollback to permanent checkpoint.
  csn ← csn value of the permanent checkpoint.
  epoch ← epoch + 1.
  send rollback_req to all processes in $(P \cup L)$.
  wait till received ack_rollback_req from
  all processes in $(P \cup L)$.
```

Leader:
(on receipt of *rollback_req* from the initiator)

```
  role ← LEADER.
  $P \leftarrow \{p : p$ is an intra-cluster process$\}$.
  $leader$ ← initiator process.

  if (in_ckpnt)
      abort checkpointing activity.
      in_ckpnt ← FALSE.
  endif
  if(msg.csn $>$ csn)
      csn $\leftarrow$ msg.csn.
      take a tentative checkpoint.
  endif
  if ((tentative checkpoint present)
    and (csn of tentative checkpoint = msg.csn))
      make it a permanent checkpoint.
  endif
  rollback to permanent checkpoint.
  epoch $\leftarrow$ epoch + 1.
  csn $\leftarrow$ csn value of the permanent checkpoint.
  send *rollback_req* to all processes in $P$.
  wait till received *ack_rollback_req* from
    all processes in $P$.
  send *ack_rollback_req* to $leader$.
```

Follower:
(on receipt of *rollback_req* from the leader)

```
  role $\leftarrow$ FOLLOWER.
  $leader$ $\leftarrow$ leader process.

  if (in_ckpnt)
      abort checkpointing activity.
      in_ckpnt $\leftarrow$ FALSE;
  endif
  epoch $\leftarrow$ epoch + 1.
  if (msg.csn $>$ csn)
      csn $\leftarrow$ msg.csn.
      take a tentative checkpoint.
  endif.
  if ((tentative checkpoint present)
    and (csn of tentative checkpoint = msg.csn))
      make tentative checkpoint permanent.
  endif.
  rollback to permanent checkpoint.
  csn $\leftarrow$ csn value of the permanent checkpoint.
  send *ack_rollback_req* to $leader$.
```

Messages with out-of-context *epoch* value are discarded at the message preprocessing layer. The message preprocessing directives of the received messages are described below:

Message Preprocessing:

```
  if (((msg=rollback_req) and (msg.epoch=(epoch+1)))
    or ((msg=ckpt_req) and (msg.csn=(csn+1))))
                                deliver(msg);
  elsif (msg.epoch$\neq$epoch) drop(msg);
  else                          deliver(msg);
  endif
```

4 Proof of Correctness

Lemma 1: The permanent checkpoints after a checkpointing session contains the same csn value.

Proof: We prove it by induction. As the base condition, the csn variable of all the processes are initialized to the same value. The csn value of a process is only incremented once at the beginning of each of the checkpoint sessions. For initial part of the proof, we assume that control messages are not lost or damaged, but can be delayed arbitrarily, and no fault occurs during a checkpoint session.

Under faultless condition, the tentative checkpoints do not live beyond a checkpoint session. Under the above assumptions all the tentative checkpoints are promoted to permanent checkpoints in the commit phase(C2) of the checkpointing protocol. By induction hypothesis, we say that at the end of n^{th} checkpointing session, the csn value of all the process is n. We have to prove that the tentative checkpoints taken in the $(n+1)^{th}$ invocation of the checkpointing protocol are marked with csn value of $(n+1)$. The initiator does not move into the commit phase(C2) of the protocol until it receives *ack_ckpt_estb* from all the follower processes in its cluster and *ack_ckpt_req* messages from the leaders of other clusters. The leaders sends *ack_ckpt_req* only when it has received *ack_ckpt_estb* from all the followers of its own cluster. Again, all the followers sends *ack_ckpt_estb* only when they have established tentative checkpoint with csn=$(n+1)$. So, at the beginning of commit phase(C2) of the initiator checkpointing protocol, all the followers and the leaders have their own tentative checkpoints with csn=$(n+1)$. Similarly, the leaders sends *ack_commit_req* only after it has received *ack_commit_req* from all its followers. The initiator terminates the commit phase(C2) only after it has received *ack_commit_req* from its followers and leaders. Therefore, at the end of $(n+1)^{th}$ checkpointing session the value of csn in all processes is $(n+1)$. Under the given assumptions it is easy to see that the protocol does not fall into *live-lock* problem.

Now we relax the assumption that control message are not lost or damaged. It can be taken care of by timeout and retransmission mechanism. However this may give rise to message duplication in the network. It is easy to see that the csn value monotonically increases under the 2^{nd} assumption. messages with unexpected csn value, or out of context messages can be easily identified and discarded at the message preprocessing layer. Therefore, relaxation of 1^{st} assumption does not affect the proof process.

Finally, we allow faults to occur during checkpoint sessions, but assume that only one fault occurs during such sessions. Observe that the whole checkpointing protocol is in line with the *2-phase commit protocol*. Fault may occur at different phases in the whole checkpointing session. Below we enumerate these cases.

Case 1. *Process fails before taking the n^{th} tentative checkpoint.*
It may so happen that some of the processes are already in the checkpoint session, and some are not. Processes, which are in the checkpointing session, never proceed to commit their $(n+1)^{th}$ tentative checkpoints, if they have taken any. This is because the coordinator never signals to commit a tentative checkpoint until it has received *ack_ckpt_estb* from all the processes that they have successfully established their $(n+1)^{th}$ tentative checkpoints. The failed process will rollback to the n^{th} permanent checkpoint

and so do the other processes. So, at the end of recovery the permanent checkpoints have the same csn value of n.

Case 2. *Process fails after establishing $(n+1)^{th}$ tentative checkpoint.*
The failed process may have failed after sending *ack_ckp_estb*, or before. In the both the cases, without loss of generality, we assume that some processes have proceeded to establish $(n+1)^{th}$ permanent checkpoint. However, by recovery protocol, the failed process will rollback to its $(n+1)^{th}$ checkpoint. Processes in the $(n+1)^{th}$ checkpoint session are already blocked. The recovery protocol achieves a complete coordination, and establishes $(n+1)^{th}$ permanent checkpoint with the non-failed processes. After completion of the recovery session all processes rollback to their $(n+1)^{th}$ checkpoint (csn value of $(n+1)$.

Case 3. *Process fails after establishing its $(n+1)^{th}$ permanent checkpoint.*
A process moves into the commit phase only after all other processes succeed in establishing their respective tentative checkpoints. Some of the processes may have already committed their $(n+1)^{th}$ tentative checkpoints, and therefore have completed their checkpoint session, can never proceed to commit another tentative checkpoint due to the failed process, in case a new checkpoint session has started. The processes which have not committed their tentative checkpoints are still in the checkpoint session. The recovery phase initiated by the failed process eventually forces processes to commit their tentative checkpoints, if they have any. Therefore, after recovery all processes go back to $(n+1)^{th}$ permanent checkpoint (csn value of $(n+1)$.

Theorem 1: The checkpoints with the same csn value form a recovery line.
Proof: The tentative checkpoints taken during a checkpoint session is consistent with each other within the boundary of a cluster, since the protocol followed is well-known coordinated protocol with the cluster leader being the coordinator. Also note that tentative checkpoints of any two processes, belonging to two different clusters, remain consistent due to the *blocking policy* which prevents creation of any new dependency among the processes. This implies that all the tentative checkpoints in a checkpoint session are consistent. Again, the checkpoints taken during a checkpoint session are marked with the same csn value, by *lemma-1*.

Theorem 2: All the processes recover from checkpoint with the same csn value.
Proof: The recovery procedure is initiated at the event of a fault and subsequent recovery of the failed process. The failed process assumes the role of *leader*. A fault may occur at any time. We enumerate different cases below and prove that after failure all the processes rolls back to a checkpoint with the same csn value.

- *Fault occurs during normal running time of the application.*
  Processes roll back to their permanent checkpoints. By lemma 1, the permanent checkpoints have the same csn value, and by theorem 1, the checkpoints with the same csn value form a recovery line.

- *Fault occurs during checkpoint phase.*
  The proof is same to that discussed in that of lemma 1, with relaxation of the 2^{nd} assumption.

4.1 Extension of the Protocol

The protocol, so far discussed, considers that there is only one checkpoint initiator. In case there are multiple concurrent initiators, each process has to handle multiple checkpoint sessions concurrently, and also maintain synchronization among them. We call one of the session a *primary checkpoint session*, when a process either initiates it or is the first checkpoint session where it gets involved. The other sessions are called *secondary checkpoint sessions*. The primary session follows the checkpoint protocol described in Section 3. It is easy to see that no process becomes ‘initiator’ in a secondary session. Since one particular process in a cluster plays the role of a leader, a ’leader’ in its primary session, can not become ‘follower’ in its secondary sessions. The activity of a process in its secondary session is to send acknowledgment messages. The acknowledgment messages indicates completion of certain checkpointing activity. For example, *ack_ckpt_req* message indicates the sender is in checkpoint mode and application processes are blocked. The message *ack_ckpt_estb* indicates that the sender has established a tentative checkpoint. However, activities of checkpoint processes are always dictated by the primary checkpoint session of a process. But the acknowledgments in its secondary session must be sent in accordance with its primary role in order to maintain consistency. For example, a process being ‘initiator’ in its primary session and ‘follower’ in its secondary session, can send *ack_ckpt_estb* to the leader only after it has taken a tentative checkpoint according to its primary session.

Recovery from multiple node failures can also be accommodated, but with slight modification to the recovery protocol. The modification is not discussed due to lack of space.

5 Results and Discussion

We have simulated the well-known coordinated checkpointing protocol (*flat*) [6] and the hierarchical checkpointing protocol (*hier*), with dPSIM [9], in a network setup where the link speed inside clusters is 10 Mbps. Simulations were carried out for the cases when link speed between two clusters is 1 Mbps and 100 Kbps. The network topology consists of 4 clusters each containing 8 computing nodes. The results are discussed below. Checkpoint overhead shown are averaged over the number of checkpoint taken. The per checkpoint overhead is calculated as the duration for which a process remains blocked due to the checkpointing protocol.

Figure 1 shows average per checkpoint overhead variation for different applications types, with increasing extra-cluster message frequency for hierarchical coordinated checkpointing protocol (*Hier*), under checkpoint interval of 100 secs. Average per checkpoint overhead of the flat coordinated protocol (*Flat*) does not show any significant variations with different application send frequencies, and therefore only one curve is shown (corresponding to send frequency of 0.5 msgs/sec). Results also show insignificant variations in checkpoint overhead for different checkpoint intervals for both the flat and the hierarchical coordinated checkpointing protocol, because per checkpoint overhead is independent of checkpoint interval. The checkpoint overhead of flat coordinated checkpointing protocol does not show any variation with increase in extra-cluster message percentage, because the protocol does not depend on the communication pattern of the application. Checkpoint overhead of the hierarchical coordinated protocol is much less than that of the flat coordinated protocol. However, checkpoint overhead of the hierarchical coordinated checkpointing protocol is sensitive to send frequency of the application process, and increases with the increase in extra-cluster message count. Because with the increase in extra-cluster message number, probability of sending an extra-cluster message increases during the second phase of the checkpointing protocol (*i.e.* during the period when policy B is enforced). As a result, a process is blocked for longer period, resulting in increase of checkpoint overhead. The checkpointing overhead of hierarchical coordinated protocol tends to that of the flat coordinated checkpointing for applications with higher send frequency and high percentage of extra cluster messages. Similar trends are visible for networks where the extra-cluster links are of 100 Kbps, and therefore is not shown.

6 Conclusion

In this paper we have presented a hierarchical checkpoint protocol, which is based on the well-known coordinated protocol. The protocol minimizes checkpointing bottleneck due to the slower links in the distributed system. We have simulated and compared both the flat and the hierarchical protocol for various environments, and results shows that hierarchical protocol suffers less checkpoint overhead than that of the flat coordinated protocol.

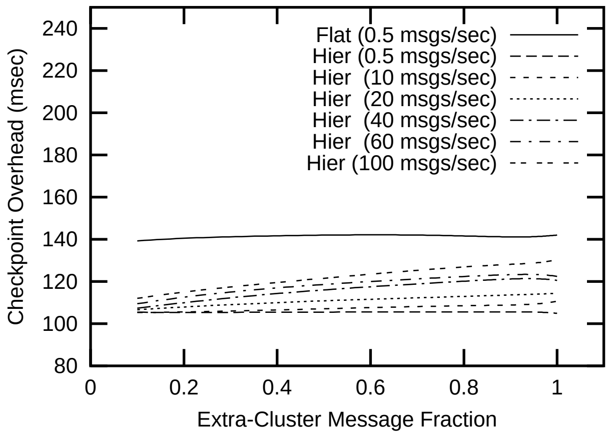

Figure 1. Checkpoint overheads under various extra-cluster message frequencies, where the extra-cluster link is 1 Mbps.

References

[1] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with transparent migration and recovery. In *3rd PVM users' Group Meeting*, May 1995.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states in distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, February 1985.

[3] E. N. Elnozahy, L. Alvisi, D. B. Johnson, and Yi-Min Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-148, School of Computer Sc, Carnegie Mellon University, June 1999.

[4] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw. Process Introspection: A heterogeneous checkpoint/restart mechanism based on automatic code modification. Technical Report CS-97-05, 25, 1997.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. Journal of Supercomputer Applications.*, 11(2):115–128, 1997.

[6] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engg.*, 13(1):23–31, 1987.

[7] J. Leon, L. Ficher, and P. Steenkiste. Fail-safe PVM: A portble package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Sc., Cernegie-Mellon University, 1993.

[8] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Symp. on Principles of Distributed Computing*, pages 121–129, 1994.

[9] H. S. Paul, A. Gupta, and R. Badrinath. Evaluation of different classes of checkpoint and recovery protocols with dPSIM. In *Proc. 4th Intl. Conf. on Information Technology (CIT), Gopalpur on Sea*, pages 315–320, 2001.

[10] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault tolerant systems. *ACM Trans. on Computer Systems*, 1(3):222–238, August 1983.