MOBILE AGENT BASED CHECKPOINTING WITH CONCURRENT INITIATIONS

PARTHA SARATHI MANDAL*
INRIA Futurs
Laboratoire de Recherche en Informatique
University Paris Sud, 91405 Orsay cedex, France
partha@lri.fr

and

KRISHNENDU MUKHOPADHYAYA
Advanced Computing and Microelectronics Unit
Indian Statistical Institute, 203, B T Road, Kolkata - 700 108, India

ABSTRACT

Traditional message passing based checkpointing and rollback recovery algorithms perform well for tightly coupled systems. In wide area distributed systems these algorithms may suffer from large overhead due to message passing delay and network traffic. Mobile agents offer an attractive option for designing checkpointing schemes for wide area distributed systems. Network topology is assumed to be arbitrary. Processes are mobile agent enabled. When a process wants to take a checkpoint, it just creates one mobile agent. Concurrent initiations by multiple processes are allowed. Synchronization and creation of a consistent global state (CGS) for checkpointing is managed by the mobile agent(s). In the worst case, for k concurrent initiations among n processes, checkpointing algorithm requires a total of $O(kn)$ hops by all the mobile agents. A mobile agent carries $O(n/k)$ (on the average) size data.

Keywords: Distributed system; fault tolerance; logical checkpoint; rollback recovery; mobile agent.

1. Introduction

Traditional checkpointing and rollback recovery algorithms work well in tightly coupled systems. But in wide area distributed systems like internet, these algorithm degrade system performance due to network traffic and message passing delays. *Mobile agents* can improve system performance for checkpointing algorithms on wide area distributed systems like internet.

A *mobile agent* is an executable program that can automatically migrate among processes as a messenger. It halts execution of the host, executes at the host process, takes necessary actions, and finally dispatches itself, together with required information, to another host. In wide area networks, a mobile agent can execute asynchronously and automatically with its own control. If a user is temporarily disconnected from the network, the agent can still carry out the tasks. Mobile agents can reduce the network congestion by reducing frequent remote communications.

A set of checkpoints, with one checkpoint for every process, is said to be a *Consistent Global checkpointing State (CGS)*, if it does not contain any *orphan message* or *missing message* [6, 13, 14]. Generation of missing messages may be acceptable, if messages are logged [1, 6, 8] by sender. However, if the processes are non deterministic, orphan messages cannot be avoided by message-logging. In case of a failure, the system rolls back to a consistent set of checkpoints and resume computation.

In the example shown in Figure 1, if a process fails after the *global state*, the system rolls back to the *global state*. With respect to the *global state*, m_3 will be an orphan message and m_2 will be a missing message. This *global state* is an *inconsistent* global state.

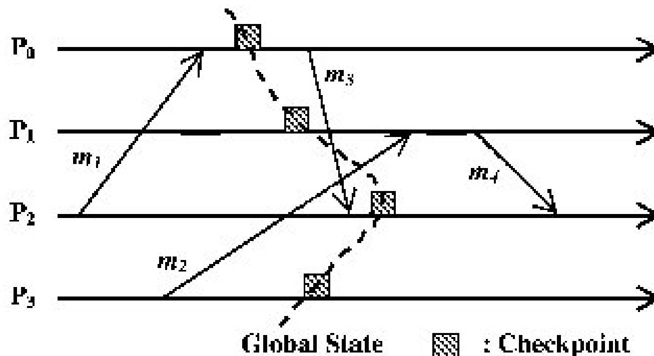

Fig. 1. An example showing orphan message (m_3) and missing message (m_2) with respect to the global state.

Checkpointing algorithms may be classified into three broad categories (a) Synchronous, (b) Asynchronous and (c) Quasi-synchronous. In synchronous [4, 5, 10, 13, 14, 15, 19, 24] checkpointing, processes synchronize their checkpointing activities through control messages, so that a globally consistent set of checkpoints is always maintained in the system. Synchronizing checkpointing activity involves message overhead. In some of the synchronizing checkpointing algorithms, all process executions may have to be suspended during the checkpointing coordination. Such algorithms are called *blocking* [9, 10] algorithms. Blocking checkpointing algorithms result in performance degradation. In *non-blocking* [4, 5, 19] algorithms application processes are not suspended when checkpoints are being taken. In asynchronous [2, 15, 23] checkpointing, processes take checkpoints without any coordination. They provides maximum autonomy for processes to take checkpoints; however, some of

the checkpoints taken may not lie on any consistent global checkpoint, thus making the checkpointing efforts *useless*. Useless checkpoints degrade system performance. Unlike uncoordinated checkpointing coordinated checkpointing does not generate useless checkpoints [7]. In asynchronous checkpointing finding a CGS can be quite tricky. The choice of checkpoints for the different processes is influenced by their mutual causal dependencies. The common approach is to use *rollback dependent graph* or *checkpoint graph* [2, 5, 6, 19, 25]. The number of useless checkpoints in asynchronous checkpointing, may be reduced by making processes take *communication induced* checkpoints besides the asynchronous checkpoints. Such checkpointing algorithms are called *quasi-synchronous* [14] checkpointing. This approach can be seen as a tradeoff between synchronous and asynchronous checkpointing.

This work describes a non-blocking synchronous checkpointing algorithm combined with communication induced checkpointing that uses intelligent mobile agents in a distributed system over an arbitrary network topology. The mobile agents intelligently move from one process to another and take checkpoints for host processes without any useless checkpoints. To visit an entire graph of n nodes, a mobile agent needs $2(n-1)$ moves along a DFS tree.

2. Related Works

Several earlier works [4, 5] on snapshot collection algorithms assume that at any point of time only one snapshot collection process is active. Koo and Toueg [10], Spezialetti and Kearns [21], Prakash and Singhal [20] and Mandal and Mukhopadhyaya [12, 13] have proposed methods for handling concurrent initiations of snapshot collection.

According to Koo and Toueg's algorithm, once a process takes a local checkpoint, either as an initiator or on request from another process, it becomes *unwilling* to take a checkpoint in response to another initiator's request. The process sends an 'unwilling' response to all subsequence requests, until the checkpoint it has taken, is made permanent or the checkpointing collection is aborted. This algorithm is blocking. Prakash and Singhal have shown that in this algorithm all the initiations may end up aborting, leading to a wastage of effort [20].

Spezialetti and Kearns (S-K) algorithm [21] forces all process to take local checkpoints similar to Chandy-Lamport checkpoint collection algorithm [4]. A process takes a local checkpoint for the first request and forwards that request to its neighbors. All subsequent requests are collected in a list called *border_list*. Once a process has received requests along all its incident edges, its checkpointing phase is complete. Then the process sends its *border_list* to the process from which it received the first checkpoint request message. In this way, mutually disjoint sets of processes take their local checkpoints in response to requests from different initiators. Finally, initiators communicate with each other and one checkpoint for each process is selected to build a CGS, which is minimal [11].

On the other hand, the Prakash and Singhal (P-S) algorithm [20] generates a CGS, which is maximal [11]. Unlike S-K algorithm the P-S algorithm permits full propagation of checkpoint requests generated by all the concurrent checkpoint initi-

ations. Thus the P-S algorithm outputs a CGS with more recent checkpoints than the S-K algorithm. For k concurrent initiators, S-K algorithm requires the transmission of $O(n^2)$ messages to take the local checkpoints and $O(k^2n)$ messages for information dissemination phase with message size $O(n/k)$. P-S algorithm requires $O(kn^2)$ messages to take tentative checkpoints. Another $O(k^2n)$ messages of size $O(n)$ are exchanged for establishing a CGS. Although the number of messages required by P-S algorithm is higher, as they are sent concurrently, the time to collect tentative checkpoints is comparable with that of S-K algorithm. The algorithm proposed by Mandal and Mukhopadhyaya [13] can handle concurrent initiations of snapshot collection for unidirectional and bidirectional rings. The worst case message and time complexities are $O(n^2)$ and $O(n)$ respectively. The message and time complexities of this proposed algorithm are both $O(n)$.

Not much is reported in the literature [3, 12], on checkpointing and recovery using mobile agents. Most of the works focus on the fault tolerance of mobile agents using checkpointing. Many fault tolerance schemes for mobile agent have been suggested, which are either replicating [18] the agents or checkpointing [16, 22] the agents. The performances of the replication scheme and the checkpointing scheme are compared in [22] and [17].

Mandal and Mukhopadhyaya (M-M) [12] presented mobile agent based communication induced checkpointing and rollback recovery algorithms for an arbitrary hamiltonian network topology. The checkpointing algorithm [12] can handle multiple concurrent initiations. Processes take logical checkpoints. Each process stores at most two checkpoints. The mobile agents take just one round along the hamiltonian cycle for the checkpointing as well the recovery algorithm. For concurrent initiations among n processes, mobile agents need $O(n^2)$ moves, in the worst case.

Cao et al [3] proposed a mobile agent enabled hybrid algorithm combining asynchronous and synchronous checkpointing. A mobile agent, called *coordinator agent*, is used to enforce that the number of checkpoints to be rolled back in case of recovery will not exceed a predefined threshold. Periodically, each process takes its local checkpoints independently according to its own needs. The coordinator agent travels from one process to another, carrying updated information of previously visited processes. Each process maintains a Message-Receive-Information Table $MRIT_i$ storing the header of the last message received from each remote process. When a process P_i takes a checkpoint, the $MRIT_i$ will be saved with the checkpoint. When a coordinator mobile agent arrives at P_i, it reads the $MRIT_i$ for all the saved checkpoints at P_i and then uses these tables to update its own *Dependency Table (DT)* that contains the necessary information about dependency between the checkpoint of P_i and those of other processes. Based on the checkpoint dependency information, the coordinator agent will calculate the number of rollbacks which need to be performed by the current process if a fault is detected at that moment. If the coordinator agent detects that the number of rollbacks exceeds the threshold value, a forced coordinated checkpointing procedure will be initiated. The coordinator agent first generates a group of *ConsistentCP* agents, one for each site, and then dispatches them to their corresponding sites. When a *ConsistentCP* agent arrives

at a remote site, it will examine the local process to check whether there are any messages sent after its last checkpoint. If so, it will request the local process to take a forced checkpoint. Before the new local checkpoint is taken, the local process is not allowed to send out any new messages. After taking a forced checkpoint, the *ConsistentCP* agent sends back a *complete_message* to Coordinator agent. The Coordinator agent will then send *start_message* to all remote sites and processes start the normal execution again.

3. System Model

We consider a distributed system consisting of n processes, one process per processor, denoted by $\{P_0, P_1, P_2, \cdots, P_{n-1}\}$ and a set of bidirectional channels on an arbitrary network topology. There is no common clock, shared memory or central coordinator. Message passing is the only mode of communication between any pair of processes. The processes are non-deterministic. Thus, orphan messages can not be handled by merely logging them. Any process can initiate checkpointing. We assume that there is no link failure, only processes may fail. The computation is asynchronous; messages are exchanged with finite but arbitrary delays.

In this paper, we consider *logical checkpoint* [15, 24], which is a standard checkpoint (i.e., snapshot of the process) plus a list of messages, which have been sent by this process but are unacknowledged at the time of taking the checkpoint. Message lists are updated continuously. Our algorithm allows the generation of missing messages in case the system has to roll back to its last checkpoint. At the time of restart after a failure, processes retransmit their unacknowledged messages (not all of whom may be missing messages). There may be duplicate messages after recovery from a failure and that is handled using message identifiers.

4. The Checkpointing Protocol

In the proposed algorithm, for each process, at most two checkpoints may have to be stored in the stable storage when checkpointing procedure is running; otherwise one checkpoint per process is enough to make a system consistent. Checkpoints have a one-bit version numbers (*v_no*). In the beginning all processes start by taking a permanent checkpoints with $v_no = 0$.

This algorithm is non-blocking, i.e., even when a checkpointing process is running, processes are free to run their applications. To avoid orphan messages, every process tags the *v_no* of its latest checkpoint with each application message header. When a process receives an application message, it first compares application message's version no (*msg_v_no*) with its own current checkpoint *v_no*. If $msg_v_no = (v_no + 1)\ mod\ 2$ and $ckpt_state = P$ then the receiver process decides that sender has taken a new checkpoint before sending the message and the checkpointing process is on. So the receiver first takes a checkpoint with $v_no = msg_v_no$. Then it sets $ckpt_state = T$ and $msg_ckpt = True$ before processing the message. Initially *msg_ckpt* is *False* for all processes.

In one checkpointing cycle, a process takes exactly one checkpoint. This checkpoint may be an induced checkpoint taken on receiving an application message with

tagged *v_no* one more than that of the process. Also it may be taken on receiving a checkpointing request, after making sure that the same has not already been taken through the effect of an application message. Once a checkpoint is taken, no further checkpoints are taken by the process on receiving application messages or checkpoint requests, till this whole checkpointing cycle is over.

Each process has a list of all its neighbors (i.e., processes which are connected with this process by direct communication links). Each process may initiate checkpointing independently. The initiator creates a mobile agent, which travels across the network and creates a CGS. The mobile agent (MA) id (*agent_id*) is the same as the process id of its creator. The mobile agent moves to other processes following an execution path of *depth-first* search, starting from its creator. The agent maintains a stack and two lists. The stack, *Stack*, has the path to the root, as required by the DFS. The list VL (Visited List) is the list of processes which have been visited by this agent before any other agent in this checkpointing cycle. All these processes would thus have taken temporary checkpoints. Note that some of them may have taken the checkpoints induced by application messages, even before this agent reached them. PLI (Partial List of concurrent Initiators) is a list of processes who have initiated checkpointing. However, this list is not the complete list of initiators. If a process that has taken a checkpoint through another agent, is visited by our agent, the initiator of the other agent is added to PLI.

4.1. *First phase*

Suppose, P_i is a checkpointing initiator. P_i creates a mobile agent with *agent_id* $= i$. The agent takes a temporary checkpoint (*ckpt_state* $= T$) and *v_no* equal to (*v_no* $+ 1$) *mod* 2. The *proc_id* of the current process, i, is pushed into the $Stack_i$. Then the agent leaves P_i and moves to a neighbor P_j.

When an agent, MA_i, visits a process P_j, it halts the application of P_j, adds its *agent_id* (i) to PLI_j. If it finds that the state of the current checkpoint of P_j is P, MA_i takes a new checkpoint for P_j with *ckpt_state* $= T$ and *v_no* $=$ (*v_no* $+ 1$) *mod* 2. If any neighbor of P_j is yet to be visited, the agent moves to that neighbor and performs the same operation. Before leaving P_j, MA_i pushes j into $Stack_i$. If all neighbors of P_j have already been visited, then MA_i moves back to the last process from which it came to P_j. The id of the last visited process is available at the top of $Stack_i$ (Top_Stack_i). Before leaving P_j, MA_i puts j in VL_i.

If P_j has already taken a temporary checkpoint (i.e., *ckpt_state* $= T$) then MA_i checks the flag *msg_ckpt*. If it is *True* then the current checkpoint was induced by an application message. MA_i sets *msg_ckpt* $=$ *False* and adds j to VL_i. If *msg_ckpt* $=$ *False*, then another agent, from a different initiator, has already visited this process. In both the cases, the information about the list of initiators available with the agent in PLI_i and the information on the same with P_j in PLI_j are merged and the updated information replaces both PLI_i and PLI_j. Finally, the agent leaves P_j and moves to either a yet unvisited neighbor of P_j (if *msg_ckpt* $=$ *True* and such a neighbor exists) or to the process from which it came.

At the end of first phase, mobile agent MA_i returns back to P_i and sets *CompletedFirstPhase* $=$ *True*. Initially the flag was *False*. The topology of the graph

is divided into clusters of processes such that processes in each cluster have taken checkpoints corresponding to one agent. The PLI of the agent, when it returns to its initiator, has the list of all the neighboring initiators. The VL of the agent, has the list of all the processes which have taken checkpoints induced by the agent.

Lemma 1. *At the end of first phase, visited lists (VL) of initiators partition the topology of the underlying graph.*

Proof. If a process is visited by one or more agents, it would be included in the VL of only the first agent to visit it. Thus the VL's are guaranteed to be disjoint. We only need to show that every process belongs to one VL. We show that every non-initiator process is visited by at least one agent.

Suppose there are processes which are not visited by any agent. Since the underlying topology is connected, there exists neighboring processes P_s and P_t such that P_s is in VL_i but P_t is not visited by any agent. Since MA_i was the first to visit P_s, when it reached P_s either P_s had no temporary checkpoint or it had a application message induced checkpoint. In the first case, MA_i is bound to explore all neighbors and hence must visit P_t too. In the second case, since MA_i is the first agent to visit P_s, it will find flag *msg_ckpt* to be *True*. In this case also, MA_i must explore all neighbors and hence visit P_t. □

Lemma 2. *At the end of first phase, every processes has one temporary checkpoint with same v_no and this set is consistent.*

Proof. Lemma 1 implies that every process has one temporary checkpoint. Since, all the processes had the same *v_no* at the beginning of the checkpointing process, all the agents will carry the identical value of *v_no*. So, the *v_nos* of the new set of checkpoints will be same too. Application message induced checkpoints guarantee that there are no orphan messages. □

Lemma 3. *At the end of first phase, $j \in PLI_i \Rightarrow i \in PLI_j$.*

Lemma 4. *In the first phase, if the number of concurrent initiations is k, the total number of moves by all the agents is $O(kn)$.*

Proof. Since an agent carries the list of visited processes, the agent travels along the edges of the DFS tree only, and an edge of the tree is traversed at most twice. So, for each agent, the number of hops traversed is $O(n)$. Thus, the total number of hops traversed by all the agents is $O(kn)$. □

It may be noted that the total number of hops cannot be claimed to be $O(n)$ as the processes visited by the different agents are not exclusive.

Algorithm: MA_i On being initiated

```
Initiator P_i
begin
   creates a mobile agent with agent_id ← i
end
```

```
MobileAgentMA_i
begin
    Push i into the Stack_i
    TakeTemporaryCheckpoint(i,i)
    while Stack_i ≠ empty do
        if neighbor (P_j) of Top_Stck_i ≠ null and
                    it is yet to be visited by MA_i then
            move to the P_j
            TakeTemporaryCheckpoint(i,j)
            if P_j is unvisited by any other agent then
                Push j into Stack_i
            end if
        else Pop Stack_i and move to Top_Stack_i
        end if
    end while
end
```

```
TakeTemporaryCheckpoint(i,j)
begin
    if msg_ckpt ≠ True ∧ ckpt_state ≠ T then
        take a new checkpoint
        ckpt_state ← T
        v_no ← (v_no − 1) mod 2
        PLI_j ← PLI_j ∪ {i}
        VL_i ← VL_i ∪ {j}
    else if msg_ckpt ≠ True ∧ ckpt_state = T then
            PLI_i ← PLI_i ∪ PLI_j
            PLI_j ← PLI_i
        else {msg_ckpt = True ∧ ckpt_state = T}
            msg_ckpt ← False
            VL_i ← VL_i ∪ {j}
            PLI_j ← PLI_j ∪ {i}
        end if
    end if
end
```

4.2. *Second phase*

In the second phase, the initiator with the minimum id generates the complete list of initiators. When an agent MA_i comes back to its initiator P_i, after the first phase, if PLI_i has an entry less than i, MA_i will remain at P_i. If all entries in PLI_i are greater than or equal to i, MA_i moves to a process, P_j in the list PLI_i.

When MA_i, reaches P_j, it pushes j into $Stack_i$. If it finds that *CompletedFirstPhase* = *False*, then it waits for the return of MA_j. After MA_j returns, PLI_i and PLI_j are merged and the new list replaces both PLI_i and PLI_j. If the new list has an entry less than i, then MA_i is destroyed. If all entries in the list are greater

than or equal to i, and if the list has a member so far not visited, MA_i moves to it. If all members have already been visited, it returns to the process it last came from. Before MA_i leaves P_j, the top of the $Stack_i$ is popped off.

Lemma 5. *After second phase, the initiator with minimum id will have the complete list of initiators.*

Proof. The agent with minimum id will not be destroyed and will go on exploring till its PLI is exhausted. Suppose we form a graph on the initiator processes, with an edge between i and j if PLI_i contains j and PLI_j contains i. (Lemma 3 ensures that, $j \in PLI_i \Rightarrow i \in PLI_j$.) This graph shall be referred to as the *reduced graph*. The movement of the minimum id agent can be thought of as a DFS on this reduced graph. So the minimum id agent will visit all the initiators and hence will come back with the complete list. □

Lemma 6. *After second phase, only the agent with minimum id will return to its creator.*

Proof. For an agent to return to its creator, it must visit all other initiators. So, if it does not have minimum id, it will visit an initiator having a lower id (or an initiator with lower value in its PLI) and hence will be destroyed. □

Lemma 7. *In the second phase the total number of moves by all the agents is $O(kn)$.*

Proof. A mobile agent travels along the tree edges of a DFS tree on the nodes it visits in the reduced graph. The DFS tree on the reduced graph can be broken down into $O(n)$ edges in the original graph. So, the number of moves for any agent is $O(n)$. Summing over k agents we have $O(kn)$ moves in total. □

Algorithm: MA_i Building the list of initiators

$\underline{MA_i \ at \ P_i}$
begin
  if $\exists$ a $j \in PLI_i$ such that $j < i$ *then*
    go to *sleep*
  else move to P_j such that $j \in PLI_i$
    Push i into $Stack_i$
  end if
end

```
MA_i at P_j
begin
   Push j into the Stack_i
   while CompletedFirstPhase = False do
      waitfor return of MA_j      {on MA_i's return}
            {set CompletedFirstPhase ← True}
   end while
   while Stack_i ≠ empty do
      UpdateTheList(i,j)
   end while
end
```

```
UpdateTheList(i,j)
begin
   PLI_i ← PLI_i ∪ PLI_j
   PLI_j ← PLI_i
   if ∃ at least one s ∈ PLI_i such that s < i then
      MA_i is destroyed
   else if ∃ a k ∈ PLI_i such that P_k is unvisited then
            move to P_k
         else Pop Stack_i and move to Top_Stack_i
         end if
   end if
end
```

4.3. *Third phase*

In the third phase, the complete list of initiators is communicated to all other initiators by the minimum id initiator. They, in turn, now send agents to the processes which took checkpoints induced by their respective agents. Suppose process P_i is the minimum id initiator. It creates two different agents. The first one visits the processes in VL_i and confirms the temporary checkpoints, deleting the old permanent checkpoints. The second agent visits the processes in PLI_i. $P_j \in PLI_i$, when visited by this second agent, activates MA_j, which confirms the temporary checkpoints for processes in VL_j. All agents return to their creators and are destroyed.

Lemma 8. *In the third phase the total number of moves for the agents is $O(n)$.*

Proof. All but one agent travel along DFS trees on disjoint parts of the network graph. The total number of moves of these agents is $O(n)$. The other agent travels along a DFS tree on the reduced graph and hence has $O(n)$ moves. □

Theorem 1. *At the end of third phase, every process has one permanent checkpoint with same v_no and this set establishes a CCS of the system.*

Proof. From lemma 2 we know that every process has one temporary checkpoint with the same *v_no* after the first phase. Lemma 5 ensures that all initiators

are visited by an agent from the minimum id initiator and lemma 1 guarantees all processes in the system confirm their checkpoints. So at the end of third phase, every process has confirmed checkpoint generated in the first phase. Since processes take checkpoints induced by application messages, the system will not have any orphan message. □

Theorem 2. *Total number of moves by all the agents in the complete checkpointing process is $O(kn)$*

Proof. The result follows from lemma 4, lemma 7, and lemma 8. □

Algorithm: On completing the full list of initiators

$\underline{MA_i \ at \ P_i}$
begin
  Push i into $Stack_i$
  MA_i move to P_j such that $j \in PLI_i$
  create a clone of MA_i
  clone move to P_k such that $k \in VL_i$
end

MA_i *at* P_j $(j \in PLI_i)$
begin
  Push j into $Stack_i$
  while $Stack_i \neq empty$ *do*
    wake up MA_j if it is asleep or recreate MA_j
    if it is destroyed
    MA_j move to P_k such that $k \in VL_j$
    if $\neg\exists k \in PLI_j$ such that P_k is unvisited *then*
      move to P_k
    else *Pop* $Stack_i$ and *move* to Top_Stack_i
    end if
  end while
end

> MA_i *at* P_j $(j \in VL_j)$
> *begin*
>   *Push* j into $Stack_i$
>   *while* $Stack_i \neq$ *empty do*
>   delete old permanent checkpoint
> $ckpt_state \leftarrow P$
>   *if* $\exists$ a $k \in VL_i$ such that P_k is unvisited *then*
>   move to P_k
>   *else* $Pop\ Stack_i$ and *move to* Top_Stack_i
>   *end if*
>   *end while*
> *end*

4.4. *An example*

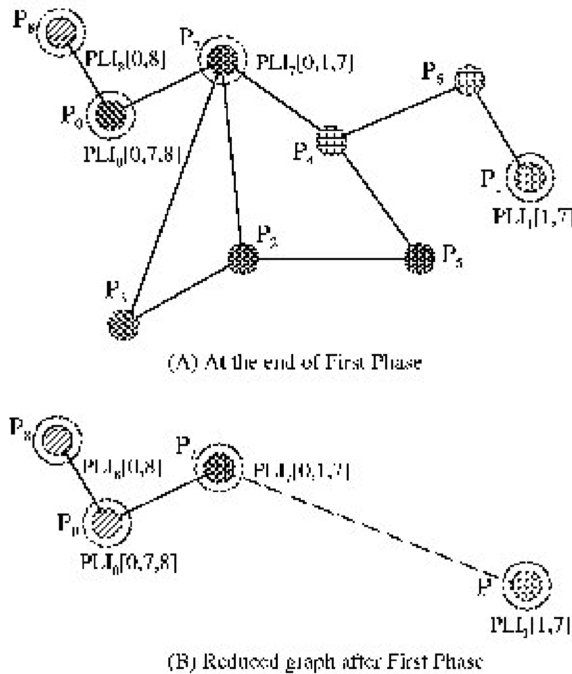

Fig. 2. An example showing multiple clusters of processes taking checkpoints for different initiators.

The different phases of the proposed scheme is illustrated with the help of an example. Figure 2 shows a system consisting of nine processes, $P_0, P_1, P_2, P_3, \cdots, P_8$. In the graph shown in the figure, a vertex represents a process and an edge represents a communication link. Processes, P_0, P_1, P_7 and P_8 initiate checkpointing concurrently and create mobile agents, MA_0, MA_1, MA_7 and MA_8 respectively. P_2, P_3, P_5 and P_7 take checkpoints induced by MA_7. P_1, P_4 and P_6 take checkpoints induced by MA_1. P_0 and P_8 take checkpoints induced by MA_0 and MA_8

respectively. Finally MA_0, MA_1, MA_7 and MA_8 return back to P_0, P_1, P_7 and P_8 with $PLI_0 = \{0, 7, 8\}$ and $VL_0 = \{0\}$, $PLI_1 = \{1, 7\}$ and $VL_1 = \{1, 4, 6\}$, $PLI_7 = \{0, 1, 7\}$ and $VL_7 = \{2, 3, 5, 7\}$, $PLI_8 = \{0, 8\}$ and $VL_8 = \{8\}$ respectively.

In the second phase, both PLI_7 and PLI_8 have id 0 in them, which is smaller than both 7 and 8. So, neither of them initiate anything in the second phase. MA_0 and MA_1 will start visiting other initiators. MA_1 visits P_7, finds that id value 0 (less than 1) is present in $PLI_7 = \{0, 1, 7\}$. Since $PLI_7 = \{0, 1, 7\}$ is a superset of $PLI_1 = \{1, 7\}$, union operation does not generate anything new for P_7. MA_1 destroys itself at P_7. MA_0 visits P_8, P_7 and P_1. During the tour, MA_0 updates PLI_0 and PLI_i at P_i for $i = 8, 7, 1$. Finally, MA_0 returns back to P_0 with the complete list of initiators, $PLI_0 = \{0, 1, 7, 8\}$.

In the third phase, MA_0 visits P_1, P_7 and P_8 along the DFS tree rooted at P_0. MA_0 confirms P_1, P_7 that PLI_1 and PLI_7 are complete, updates $PLI_8 = \{0, 8\}$ to $PLI_8 = \{0, 1, 7, 8\}$ and confirms. On getting confirmation, MA_0 visits P_0, MA_1 moves to P_1, P_6 and P_4; MA_7 moves to P_2, P_3, P_5, and P_7, and MA_8 visits P_8. At every process, the visiting mobile agent deletes the old permanent checkpoint, changes the current checkpointing state from temporary to permanent, returns back to the respective initiator and is destroyed. Finally, the system has a CGS with one checkpoint per process with same *c_no*.

5. Comparison with Existing Algorithms

In this Section, the proposed scheme is compared with the existing message passing protocols where multiple processes initiate checkpointing concurrently [20, 21]. The metric for message complexity is the total number of hops traversed by all the messages. In the proposed scheme, one hop movement by the agent is considered one hop and total number of hops moved by all the agents is the measure for total message complexity.

This is a three phase algorithm similar to the two phase algorithm of Spezialetti-Kearns [21] and Prakash-Singhal algorithm [20]. Table 1 compares of the proposed algorithm with the Spezialetti-Kearns, the Prakash-Singhal, and the Mandal-Mukhopadhyaya [12] algorithm. In the first phase, the mobile agent created by the checkpointing initiator visits other processes along a DFS tree rooted at its creator, instead of diffusing the checkpoint request, as proposed by Spezialetti-Kearns. For k concurrent initiators, Spezialetti-Kearns algorithm requires $O(n^2)$ messages and Prakash-Singhal algorithm requires $O(kn^2)$ messages to take tentative checkpoints in first phase. For the same number of initiators, in the first phase of the proposed algorithm, the total number of moves by all the agents is $O(kn)$.

In the information dissemination phase (i.e., second phase) of Spezialetti and Kearns algorithm, the initiators communicate with other initiators several times. For k concurrent initiators among n processes, $O(k^2n)$ control messages, of size $O(n/k)$ each, are required. In the second phase of Prakash and Singhal algorithm, $O(k^2n)$ messages of size $O(n)$ are exchanged for establishing a CGS. While in the second phase of the proposed algorithm; not all agents move. The agents which move, follow the same DFS tree like the first phase. The number of hops traversed

Table 1. Performance of the proposed algorithm and other existing message-passing algorithms.

	S-K [21]	P-S [20]	M-M [12]	Proposed Algorithm
Network topology	General	General	Hamiltonian	General
Worst case messages/ moves complexity	$O(n^3)$ messages	$O(n^3)$ messages	$O(n^2)$ moves	$O(n^2)$ moves
Time complexity	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Control message size for k concurrent initiations	$O(n/k)$ (average)	$O(n)$	$O(1)$	$O(n/k)$ (average)
No. of checkpoints (each process) stored for k concurrent initiations	one permanent, one temporary	one permanent, k temporary	one permanent, one temporary	one permanent, one temporary
Maximum no. of checkpoints rollback after a failure	one temporary checkpoint	k temporary checkpoints	one temporary checkpoint	one temporary checkpoint

Table 2. Performance of the proposed algorithm and other existing agent-based algorithms.

	Network topology	Algorithm	Data carried by agents	Checkpoints stored by a process	No. of rollbacks
Cao et al [3]	General	Hybrid & blocking	$O(n^2t)$	t	t
Proposed Algorithm	General	Synchronous & non blocking	$O(n/k)$	one permanent one temporary	at most one checkpoint

by an agent is $O(n)$. For k concurrent initiations, the complexity is $O(kn)$. The second phase converges in (n) time. The third phase of our algorithm is also $O(n)$.

A mobile agent in the algorithm proposed by Cao et al [3] carries the matrix DT of size $n \times t$ (where t — threshold value). Each element of DT is a list of length $(n-1)$. So the mobile agent carries $O(n^2 t)$ data. Each process has to store t checkpoints in stable storage. A mobile agent of the proposed algorithm carries $O(n/k)$ data (on the average) and stores at most two checkpoints for k concurrent initiators. One permanent and one temporary checkpoints are stored when the checkpointing algorithm is running. Otherwise one permanent checkpoint is sufficient for complete recovery. Table 2 compares of the proposed checkpointing algorithm with Cao et al [3] algorithm.

6. Conclusion

In this paper, we have proposed a mobile agent-based checkpointing protocol for arbitrary network topology. Processes take logical checkpoints. The protocol can handle multiple initiations of checkpointing. At most two-checkpoints (one permanent and other temporary) have to be saved in the stable storage of a process. Each initiator creates a mobile agent. An agent moves along a DFS tree rooted at the creator of the agent. In worst case a total of $O(n^2)$ moves and $O(n)$ time are required by all the agents in the complete checkpointing process. This is an improvement over $O(n^3)$ messages in Spezialetti-Kearns as well as Prakash-Singhal algorithms. The Mandal-Mukhopadhyaya algorithm also has $O(n^2)$ moves, but that works only for Hamiltonian topologies.

References

1. L. Alvisi, K. Bhatia and K. Marzullo, Causality tracking in causal message-logging protocols, *Distributed Computing*, 15(1): 1-15, 2002.
2. B. Bhargava and S. R. Lian, Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach, in *Proc. 7th IEEE Symp. Reliable Distributed Syst.*, 3-12, 1988.
3. J. Cao, G. H. Chan, T. S. Dillon and W. Jia, Checkpointing and rollback of wide-area distributed applications using mobile agents, in *Proc. IEEE Int. Parallel and Distributed Processing Symp.*, USA, 2001.
4. K. M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, *ACM Trans. on Computer Systems*, 3(1): 63-75, 1985.
5. G. Cao and M. Singhal, On coordinated checkpointing in distributed systems," *IEEE Trans. Parallel and Distributed Syst.*, 9(12): 1213-1225, 1998.
6. E. N. Elnozahy, L. Alvisi, Y. M. Wang and D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys*, 34(3): 375-408, 2002.
7. J. M. Hlary, A. Mostefaoui, R. H. B. Netzer and M. Raynal, Communication-based prevention of useless checkpoints in distributed computations, *Distributed Computing*, 13(1): 29-43, 2000.
8. D. Johnson and W. Zwaenepoel, Recovery in distributed systems using optimistic message logging and checkpointing, *J. of Algorithms*, 3(11): 462-491, 1990.

9. J. L. Kin and T. Park, An efficient protocol for checkpointing recovery in distributed system, *IEEE Trans. Parallel and Distributed Syst.*, 5(8): 955-960, 1998.
10. R. Koo and S. Toueg, Checkpointing and rollback-recovery for distributed system, *IEEE Trans. Software Eng.*, 13(1): 23-31, 1987.
11. F. Mattern, Virtual Time and Global States of Distributed Systems, in *Proc. of the Workshop on Parallel and Distributed Algorithm*, pp. 215-226. Elsevier Science Publishers B. V., 1989.
12. P. S. Mandal and K. Mukhopadhyaya, Checkpointing and recovery algorithms using mobile agents on a hamiltonian topology, in *Proc. of 6th Int. Conf. on High Performance Computing in Asia Pacific Region (HPC-2002)*, India, pp. 492-499, Dec. 2002.
13. P. S. Mandal and K. Mukhopadhyaya, Concurrent checkpoint initiation and recovery algorithms on asynchronous ring network, *Journal of Parallel and Distributed Computing (Elsevier)*, 64(5): 649-661, 2004.
14. D. Manivannan and M. Singhal, Quasi-synchronous checkpointing: Models, characterization, and classification, *IEEE Trans. Parallel and Distributed Syst.*, 10(7): 703-713, 1999.
15. K. Z. Meth and W. G. Tuel, Parallel checkpoint/restart without message logging, in *Proc. IEEE 28th Int. Conf. on Parallel Processing*, 253-258, 2000.
16. T. Osman, W. Wagealla and A. Bargiela, An approach to rollback recovery of collaborating mobile agents, *IEEE Trans. on Systems, Man and Cybernetics-Part C: Applications and Reviews*, 34(1): 48-57, 2004.
17. T. Park, I. Byun, H. Kim and H. Y. Yeom, The Performance of Checkpointing and Replication Schemes for Fault Tolerant Mobile Agent Systems, in *Proc. 21th IEEE Symp. on Reliable Distributed Syst.*, 2002.
18. S. Pleisch and A. Schiper, FATOMAS - A fault-tolerant mobile agent system based on the agent-dependent approach, in *Proc. Int. Conf. on Dependable Systems and networks*, 215-224, 2001.
19. R. Prakash and M. Singhal, Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems, *IEEE Trans. Parallel and Distributed Syst.*, vol. 7, no. 10, pp. 1035–1048, Oct. 1996.
20. R. Prakash and M. Singhal, Maximal global snapshot with concurrent initiators, in *Proc. Sixth IEEE Symp. Parallel and Distributed Processing*, 334-351, 1994.
21. M. Spezialetti and P. Kearns, Efficient distributed snapshots, in *Proc. of the 6th Int. Conf. on Distributed Computing Syst.*, 382-388, 1986.
22. M. Strasser and K. Rothermel, System Mechanism for Partial Rollback of Mobile Agent Execution, in *Proc. 20th Int. Conf. on Distributed Computing Syst.*, 2000.
23. R. E. Strom and S. Yemini, Optimistic recovery in distributed systems, *ACM Trans. on Computer Syst.* 3(3): 204-226, 1985.
24. N. H. Vaidya, Staggered consistent checkpointing, *IEEE Trans. Parallel and Distributed Syst.*, 10(7): 694-702, 1999.
25. Y.-M. Wang, Consistent global checkpoints that contain a given set of local checkpoints, *IEEE Trans. Computers*, 46(4): 456-468, 1997.