An Efficient Checkpointing Protocol for Mobile Distributed Systems

Raman Kumar, Parveen Kumar
Research Scholar, Deptt. of Computer Sci. & Engg., Mewar University, Chittorgarh, Rajesthan, INDIA
Deptt. of Computer Science & Engineering
rmn.kmr1@gmail.com, pk223475@gmail.com

Abstract: A mobile computing system consists of mobile and stationary nodes, connected to each other by communication network. The system raises several constraints such as limited battery life, mobility, disconnection of hosts and lack of stable storage. To reduce the lost of computational work during recovery from the node failures periodic collection of a consistent snapshot of the system (checkpoint) is required. This paper presents an efficient coordinated checkpoint protocol which is non-blocking and not forces every node to take local checkpoint. We proposed that collected global snapshot is consistent. Our protocol meet the low energy consumption, reduces storage overhead having low communication and low band width constraints of mobile computing systems.

Keywords: Mobile Computing Systems, Coordinated checkpointing, Consistent Checkpoints, Global Snapshot, Recovery.

I. INTRODUCTION
A mobile computing system is a distributed system where some of the nodes are mobile computers (Mobile Hosts (MHs)) [9]. As time passes mobile computers location gets change. To communicate with MHs, mobile support stations (MSSs) are added. An MSS communicate with other MSS by wired networks and with MHs with wireless network. Each of saved state is called snapshot (checkpoint). All the processes in the system take their checkpoints periodically.

The checkpointing techniques do not require user interaction and can be classified into following categories: (a) Uncoordinated checkpointing (b) Coordinated checkpointing (c) Quasi-Synchronous (d) Message – Login based checkpointing [14]. In this paper we concentrate on coordinated checkpointing technique which maintains a consistent snapshot of system all the times. A consistent global snapshot indicates set of N local snapshots (checkpoints) one from each process forming a consistent system state which can be used to restart process execution upon a failure. It is desirable to minimize the amount of lost work by restoring the system to most recent consistent global checkpoint. A good snapshot collection algorithm should be Non-Blocking i.e. which does not force the nodes in the system to stop their computations during snapshot collection. An efficient algorithm keeps minimum effort required for collecting a consistent snapshot to a minimum. The snapshot collection algorithm by Chandy and Lamport forces every node to take its local snapshots but the computation is allowed to continue while the global snapshot is being collected [1]. In Koo and Toueg’s algorithm all the nodes are not forced to take their local snapshots [7]. However, the underlying computation is suspended during snapshot collection. We propose a new coordinated checkpoint protocol which is non-blocking and efficient that forces a minimal set of nodes to take their snapshot and underlying computation is not suspended during snapshot collection.

II. RELATED WORK
In Chandy-Lamport algorithm [1] control messages are sent to all the nodes for consistent global checkpoint. Hence message send overhead is increased along all the channels of network.

Acharya-Badrinath algorithm [9] proposed an uncoordinated checkpointing algorithm for mobile distributed system because they found the limitations of high cost to receive request messages along every channel in network and absence of local checkpoint of MH during disconnect interval in coordinated checkpoint algorithm.

In Koo-Toueg Algorithm [7] the underlying computation is blocked. There is direct dependency approach is used while global snapshot collection. Such algorithm is not suitable for concurrent initiation.

In Venkatassan and Juang’s optimistic failure recovery algorithm [15] no dependency information is send with the computation messages. Hence while recovery process too many rollback occurs.

In [3] Guohong Cao and Mukesh Singhal had proposed an efficient algorithm that neither forces all the processes to take checkpoints nor blocks the underlying computation during checkpointing and which significantly reduces the number of checkpoints. In this paper it is described that there does not exist a nonblocking algorithm that forces only a minimum number of processes to take checkpoints. Their algorithm requires minimum number of processes to take tentative checkpoints and thus minimizes the workload on stable storage server. Their algorithm has three kinds of checkpoints: tentative, permanent and forced. Tentative and permanent checkpoints are saved on stable storage.
Forced checkpoints do not need to be saved on stable storage. They can be saved on any where even in the main memory. When a process takes a tentative checkpoint, it forces all dependent processes to take checkpoints. However a process taking a forced checkpoint does not require its dependent processes to take checkpoint. Thus taking a forced checkpoint avoids the cost of transferring large amount of data to stable storage.

**III. SYSTEM MODEL**

A message passing system consists of \( N \) fixed number of nodes that communicate each other only through messages[Fig. 1].

![Message Passing System](image)

The messages generated by underlying distributed application will be referred to as computation messages. Messages generated by the nodes to advance checkpoints, handle failures and for recovery will be referred to as system messages [Fig. 2]. In this paper the horizontal lines extending towards right hand side represent the execution of each process (MH) and arrows between them represent the messages. Processes have access to a stable storage device that survives failures. The number of tolerated process failures may vary from 1 to \( N \) [14].

**A. Propagation of minimal dependency information**

The dependency is created by means of messages between nodes. Node \( P_i \) maintains a Boolean vector \( R_i \) of \( n \) components. At \( P_i \), the vector initialized as follows:

\[
R_i[j] = \begin{cases} 
1 & i = j \\
0 & i \neq j 
\end{cases}
\]

When a node \( P_i \) sends a message to \( P_j \) it then modifies vector \( R_i \). This informs \( P_j \) about the nodes that have affected \( P_i \).

While processing a message \( M \) \( P_j \) extracts Boolean vector \( M.R \) from the message and uses it to update \( R_j \) as follows: \( R_j[k] \leftarrow R_j[k] \lor M.R[k] \), where \( 1 \leq k \leq n \).

Following diagram shows the dependency information through messages: Since \( P_2 \) was dependent on \( P_1 \) before sending \( M_2 \) to \( P_3 \); \( P_3 \) becomes transitively dependent on \( P_1 \) on receiving \( M_2 \) [Fig. 3].

![Propagation of dependency information via \( R[i] \) vector](image)

The dependency information is used to minimize the effort required to collect global checkpoint. But there should be avoidance of useless checkpoint in global checkpoint collection. The following figure describes : There are three processes \( P, Q \) and \( R \). Let \( Q \) initiates checkpoint request to processes \( P \) and \( R \). Let \( P \) and \( R \) take their local checkpoints. If \( R \) sends message \( M \) to \( P \) before receiving checkpoint request then message \( M \) will become an orphan message which creates a problem during snapshot collection. To avoid such problem a concept of checkpoint sequence number get arise. We call this ckpt_num in our protocol.

Let us describe about the global recovery line by an example [Fig. 4]. The following diagram shows the vertical line \( G_i \) the global...
Let a Mobile Host MH be initially connected to MSS.

After that MSS disconnects checkpoint at the beginning of the computation. Let Process $P_1$ initiates a new snapshot collection. Only $P_1$ and $P_2$ need to take their local snapshot because they depends upon node $P_2$. But the nodes $P_1$ and $P_2$ need not take their snapshot because they do not have dependencies on to process $P_3$. Where dotted line $G_2$ shows the global recovery line or current global checkpoint.

**B. Managing Node Mobility and Disconnection Title**

Let a Mobile Host MH be initially connected to MSS. It disconnects from MSS; after a finite period of time it connects with MSS. In such disconnected period: (a) only local events can take place on MH. (b) There is no message arrive or send events occur during this interval. Hence no any dependency events with respect to another node are created during this interval.

**Disconnection**

Disconnection of an MH is a voluntary operation and it may take arbitrary period of time. At the time of disconnection from MSS:

(a) MH takes its local checkpoint which is stored at MSS as disconnect(checkpoint), which serves request messages for MH to take checkpoint

(b) Stores its dependency vector $R$ at MSS.

(c) The Computation messages, for MH arriving at MSS during disconnect interval are stored at MSS until the end of the interval.

(d) Self identity at its stable storage at MSS.

**Reconnection**

At the time of reconnection to MSS: MH executes a reconnection protocol. The reconnection protocol sends a message through MSS to MSS. On receiving the message MSS executes the following steps:

(1) If MSS had processed request message for MH then disconnect(checkpoint), and the buffered messages are sent to MH.

(2) If no checkpoint request for MH was received by MSS during disconnect interval only buffered messages are sent.

(3) After that MSS removes the buffered messages, disconnect(checkpoint), and MH’s dependency vector. When the data sent by MSS arrives at MH, MH executes the following actions:

(a) If the received data contains disconnect(checkpoint), MH stores this checkpoint as its local checkpoint and resets all except the ith component of dependency vector $R$, before processing the messages.

(2) Process all the received buffered messages.

(3) The dependency vector is updated.

Now this reconnect protocol ends and MH makes normal communication.

MSS removes the disconnect_checkpoint, at the end of disconnect interval. In such a way mobility and disconnection of MH get managed.

**V. MINIMAL CHECKPOINTING PROTOCOL**

In this section, we present a nonblocking snapshot collection protocol for mobile distributed system. The protocol forces a minimum set of nodes to take local checkpoints. Thus overhead of checkpoint collection get minimized. After the coordinated snapshot collection terminates, the nodes that did not participate in snapshot collection can take their local checkpoints in lazy phase approach. When a node initiates a request for snapshot collection to another node then that node takes its local snapshot and propagating the request to neighbouring nodes. A global snapshot is collection of all the local nodes which participates for snapshot initiation. The snapshot thus generated is latest than each of the snapshot thus collected independently. Thus amount of lost work during rollback, after the node failure is minimized. The underlying computation need not to be suspended during snapshot collection.

**A. Data structures**

$R_i$: a Boolean vector $R_i$ of $n$ components. At $P_i$, the vector initialized as follows: $R_i[1] = 1; R_i[j] = 0$ if $i \neq j$.

When a node $P_i$ sends a message to $P_j$, it then changes vector $R_i$. This tells $P_j$ about the nodes that are dependent on $P_i$. While processing a message $M P_j$ extracts Boolean vector $M.R$ from the message and uses it to update $R_j$ as follows: $R_{j[k]} \leftarrow R_{j[k]} \text{OR} M.R[k]$, where $1 \leq k \leq n$.

$\text{ckpt_num}$: when the node takes its local checkpoint then this integer number is increased.

$\text{weight}$: A nonnegative real variable with maximum value 1 used to detect the termination of snapshot collection or checkpointing algorithm.

$\text{transmit}$: a Boolean array of size $n$ maintained by each node in its stable storage. This array is initialized to all zeros. It is used to keep the trail of those nodes to which checkpoint requests were sent by node. If in this array each element has all 0s then response message is sent to the snapshot initiator with a weight equal to weight received in the request. If in this array some elements are put to 1 then for all i such that transmit[i] = 1, a request is sent to $P_i$ with a non zero segment of weight received in request message and rest part of weight is sent to initiator with a response message.

$\text{trigger}$: A set of 2-tuples (init_id, init_ckptnum) maintained by each node, where init_id indicates the identifier of checkpointing initiator. Where init_ckptnum shows the checkpoint number of the initiator node when it took its own local snapshot on initiating the snapshot collection. trigger is changed for all system messages and the first computation message that a node sends to every other node after taking a local snapshot.

$\text{ckpt_array}$: This is an array of $n$ integer maintained at each node, ckpt_array[i] indicates the ckpt_num of the next message expected from node $P_i$.

$\text{self_trigger}$: The trigger tuple of a node receiving computation message

$\text{msg_trigger}$: Trigger tuple of computation message

$\text{get_weight}$: The weight received by dependent nodes
**forward_weight**: The weight sends by the node which further spread checkpoint request.

**B. The Protocol**

**Checkpoint initiation process**

Let \( P_i \) be the checkpoint initiator. It takes following action:

1. It takes a tentative local checkpoint.
2. Increments its \( ckpt\_num \) (3) initialized weight to 1
3. It sets \( init\_id \) and \( init\_ckptnum \) in its trigger tuple
4. It sends checkpoint request message to all its dependent nodes. The request message now includes: weight, initiator’s trigger and dependency vector \( R_i \).

**Response of a node receiving of checkpoint request**

Let node \( P_j \) receives checkpoint request.

if \( (reqst\_msg.trigger \neq P_i.trigger) \) then

\[
P_j \text{ takes tentative local checkpoint;}
\]
\[
P_j \text{ propagates reqst\_msg to all the dependent nodes but not}
\text{M.R; // Explained in further Checkpoint..... module //}
\]
\[
\text{Send portion of the received weight with its reqst\_msg;}
\]
\[
\text{Update initiator trigger tuple.}
\]
\[
\text{Send response\_msg to the initiator.}
\]

else

\[
P_j \text{ does not take the local checkpoint}
\]
\[
\text{if (transmit[i] = 0) then}
\]
\[
\text{P_j send response\_msg with weight received with}
\text{reqst\_msg to initiator.}
\]

else

\[
P_j \text{ send reqst\_msg to nodes for which transmit[j] =1}
\]
\[
\text{with remaining weight to initiator.}
\]

**Response of a node receiving of computational message**

Let a node \( P_j \) receives a computation message \( M \) from other node \( P_i \), then following action occurs:

If \( (ckpt\_num_i \leq ckpt\_array[i]) \) then

\[
P_j \text{ will not take any checkpoint;}
\]
\[
\text{Restart the computation by processing message M;}
\]

else

\[
\text{P_j has already taken a checkpoint before sending M and}
\text{this is the first computation message sent from P_i to P_j.}
\]
\[
\text{M carries a trigger (init\_id, init\_ckptnum)//}
\]
\[
\text{Set \( ckpt\_array[i] = ckpt\_num_i \).}
\]

If \( (msg\_trigger = self\_trigger) \) then

\[
\text{// P_j and P_i has taken checkpoints w.r.t same initiator//}
\]
\[
\text{update dependency vector R[i];}
\]

else

\[
\text{If (msg\_trigger.pid \neq self\_trigger.pid)}
\]
\[
\text{If (P_j had processed a message from node P_i) then}
\]
\[
\text{Return;}
\]
\[
P_j \text{ takes tentative checkpoint;}
\]
\[
\text{Set msg\_trigger=self\_trigger;}
\]
\[
\text{Propagate snapshot request to dependent processes;}
\]

**Further Checkpoint request propagation**

Let node \( P_i \) take checkpoint request. It propagates checkpoint request to its dependent processes as follows:

Take local checkpoint;

Update \( ckpt\_num \) and transmit[i];

Self\_trigger=msg\_trigger;

\[
\text{transmit[i] = R_i - M.R; for all k dependent nodes set transmit[k]=1}
\]
\[
\text{get_weight = get_weight/2}
\]
\[
\text{forward\_weight = get_weight; // Send following module to dependent nodes //}
\]
\[
\text{send(P_i.request\_msg,ckpt\_num,self\_trigger,forward\_weight);}\]

**Closing Checkpoint collection**

When the initiator receives weights from all the response messages then initiator makes the addition of all the weight when this addition becomes equal to 1. It decides that all the nodes involved in snapshot collection have taken local checkpoints. Then it propagates the commit message to all those nodes. The previous permanent local checkpoints at these nodes are discarded. Now if further recovery is required the nodes will rollback to current checkpoint.

**C. Example**

Following example clarifies the concepts used in our algorithm with the help of Fig node \( P_1 \) initiates snapshot collection by taking its local checkpoint. The node \( P_2 \) and \( P_4 \) shows dependencies to \( P_3 \). The broken arrows shows request messages sent to \( P_2 \) and \( P_4 \) to take their snapshots on their timeline. \( P_i \) takes first snapshot and then sends a message \( M_3 \) to \( P_2 \). When \( M_3 \) reaches \( P_2 \) it is the first message reached at \( P_2 \) such that \( msg\_trigger.pid \neq self\_trigger.pid \). Hence \( P_2 \) takes its snapshot before processing \( M_3 \). Node \( P_2 \) takes its local independent snapshot before sending a message \( M_4 \) to \( P_2 \).

**Fig. 5 Global Snapshot collection**

The interval number of \( M_4 \) is greater than the value expected by \( P_4 \) from \( P_1 \) [Fig. 5]. But when \( M_4 \) reaches \( P_2 \) it is not the first computation message received by \( P_2 \) with a higher interval number than expected whose \( msg\_trigger.pid \) is different from \( P_1 \)’s \( self\_trigger.pid \). So a snapshot is not taken because it will create inconsistency: The reception of \( M_4 \) will be recorded if \( P_2 \) takes a snapshot just before it processes \( M_4 \), but the transmission
of M3 will not have been recorded by P1 and now M3 becomes orphan. Also here msg\(_{\text{trigger}}\) of M3 = self\(_{\text{trigger}}\) of request message to P2. Hence no need to take further checkpoint.

VI. CONCLUSIONS

An efficient recovery mechanism for mobile computing system is required to maintain the continuity of computation in the event of node failures. In this paper we have proposed low-overhead checkpoint collection protocol to meet requirements of node mobility, energy conservation and low communication bandwidth. Dependency information among nodes is used to advance the global checkpoint of the system in coordinated manner. The proposed snapshot collection protocol is Non-Blocking i.e. the participating node does not require to stop their computation during snapshot collection. What actions are carried out when a MH disconnects from MSS and its reconnection to MSS are presented in our paper.

REFERENCES


[5]. Weigang Ni, Susan V. Vrbsky and Sibabrata Ray “Pitfalls in Distributed Non blocking Checkpointing”, University of Alabama


