Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems

Ravi Prakash and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210.
e-mail: {prakash, singhal}@cis.ohio-state.edu

Abstract

A mobile computing system consists of mobile and stationary nodes, connected to each other by a communication network. The presence of mobile nodes in the system places constraints on the permissible energy consumption and available communication bandwidth. To minimize the lost computation during recovery from node failures, periodic collection of a consistent snapshot of the system (checkpoint) is required. Locating the mobile nodes contributes to the checkpointing and recovery costs. Synchronous snapshot collection algorithms, designed for static networks, either force every node in the system to take a new local snapshot, or block the underlying computation during snapshot collection. Hence, they are not suitable for mobile computing systems. This paper presents a synchronous snapshot collection algorithm for mobile systems that neither forces every node to take a local snapshot, nor blocks the underlying computation during snapshot collection. If a node initiates snapshot collection, local snapshots of only those nodes that have directly or transitively affected the initiator since their last snapshots need to be taken. We also propose a minimal rollback/recovery algorithm in which the computation at a node is rolled back only if it depends on operations that have been undone due to the failure of node(s). Both the algorithms have low communication and storage overheads, and satisfy the low energy consumption and low bandwidth constraints of mobile computing systems.

Key words : checkpointing, causal dependency, global snapshot, mobile computing systems, portable computers, recovery.

1 Introduction

A mobile computing system is a distributed system where some of nodes are mobile computers [3]. The location of mobile computers in the network may change with time. The fixed nodes in the system are connected by a static network. A mobile node communicates with the other nodes in the system through a fixed node to which it is connected. The nodes have no common clock and no shared memory among them. They communicate with each other through messages. Each node operates independently of the others, with occasional asynchronous message communication.

In this paper we concentrate on the checkpointing and recovery aspects of mobile computing systems. In synchronous checkpointing algorithms, a consistent snapshot of the system (also called a checkpoint) is maintained at all times. In asynchronous algorithms, the constituent nodes take their local snapshots independently, and a local snapshot is selected for each node to construct a consistent snapshot of the system at the time of recovery. A consistent global snapshot indicates a *possible* state of the system if the local states of all the nodes and the messages in transit along all the channels are recorded simultaneously. In a consistent global snapshot, the reception of a message is recorded by a node only if the corresponding send has been recorded. If a node fails, the system is rolled-back to the latest consistent global snapshot [12, 19, 20, 23, 24], and then the computation proceeds from that point onwards.

To minimize the lost computation during recovery from node failures, periodic collection of a consistent snapshot of the system, to advance the checkpoint, is required. Thus, collection of a consistent snapshot of a mobile system is an important issue in the recovery from node failures. A good snapshot collection algorithm should be *non-intrusive* and *efficient*. A non-intrusive algorithm does not force the nodes in the system to freeze their computations during snapshot collection. An efficient algorithm keeps the effort required for collecting a consistent snapshot to a minimum. This can be achieved by forcing a minimal subset of nodes to take their local snapshots, and by employing data-structures that impose low memory overheads. Consistent snapshot collection algorithms for static distributed systems have been proposed in [6, 7, 15, 12, 13, 16, 17]. The snapshot collection algorithm by Chandy and Lamport [6] forces every node to take its local snapshot. The underlying computation is allowed to proceed while the global snapshot is being collected. Snapshot collection algorithms in [7, 13, 16, 17] also force every node to take its snapshot. In Koo-Toueg's algorithm [12], all the nodes are not forced to take their local snapshots. However, the underlying computation is suspended during snapshot collection. This imposes high run-time overheads on the system. Manetho [7] employs a snapshot algorithm similar to Koo-Toueg's without suspending the underlying computation. However, *all* the nodes are forced to take their local snapshots.

The mobility of nodes in the system raises some new issues pertinent to the design of checkpointing and recovery algorithms: locating nodes that have to take their snapshots, energy consumption constrains, and low bandwidth for communication with the mobile nodes. We propose a new synchronous snapshot collection algorithm that accounts for the mobility of the nodes and addresses these issues. The algorithm forces a minimal set of nodes to take their snapshots, and the underlying computation is not suspended during snapshot collection. As a result, the algorithm is non-intrusive as well as efficient. It imposes low run-time overheads on the memory and the communication network. An interesting part of the algorithm is that it has a *lazy phase* that enables nodes to take local snapshots in a quasi-asynchronous fashion, after the coordinated snapshot collection phase (the *aggressive phase*) is over. This further reduces the amount of computation that is rolled back during recovery from node failures. Moreover, the *lazy phase* advances the checkpoint slowly, rather than in a burst. This avoids contention for the low bandwidth channels.

In previous recovery algorithms for static distributed systems, such as [19], the computation at *all* the nodes is rolled back to a mutually consistent state during recovery. In [7, 23], no non-faulty node is made to roll-back its computation. However, [23] requires extensive logging of message contents, at the sender as well as receiver ends. In [7] the antecedence graph, containing the entire causal relationship, is kept in volatile storage and periodically copied to stable storage. Each computation message has to carry portions of the antecedence graph, significantly increasing the size of the messages. This can be justified for systems where node roll-backs are very expensive or are impossible (i.e., real-time systems). These mechanisms require large storage at the nodes and high bandwidth channels, both of which are in conflict with the low available bandwidth and low energy consumption goal of mobile computing systems. We propose a recovery algorithm that requires a minimal number of nodes to undo their computations on node failures and has modest memory and communication overheads. The algorithm also copes with the changing topology of the network due to the mobility of the nodes.

The key to both the algorithms are the inter-node dependencies created by messages. Specifically, there is a dependency from the sender of a message to the receiver. The inter-node dependencies considered in the rest of the paper capture the *happened before* relationship described in [14]. During a snapshot collection, only the nodes from which there is a dependency onto the snapshot initiator, either direct or transitive, since their last checkpoints, are made to take their snapshots. During recovery, only those nodes whose states are dependent on the undone operations of the failed node are made to roll-back.

Rest of the paper is organized as follows: Section 2 presents the system model. In Section 3, we discuss the issues pertinent to snapshot collection of mobile computing systems, and data structures required to keep track of the minimal dependency information at each node. Section 4 presents a non-blocking distributed snapshot collection algorithm. Section 5 presents a strategy to recover from node-failures. Section 6 compares the algorithms proposed in this paper with the existing algorithms. Finally, Section 7 presents conclusions.

2 System Model

The system is composed of a set of n nodes, and a network of communication links connecting the nodes. Some of the nodes may change their location with time. They will be referred to as *mobile hosts* or MH [1, 3]. The static nodes (henceforth, referred to as mobile support stations or MSS [1, 3]) are connected to each other by a static network. An MH can be directly connected to at most one MSS at any given time and can communicate with other MHs and MSSs only through the MSS to which it is directly connected. The links in the static network support FIFO message communication. As long as an MH is connected to an MSS, the channel between them also ensures FIFO communication in both the directions. Message transmission through these links takes an unpredictable, but finite amount of time. During normal operation, no messages are lost or modified in transit. The system does not have any shared memory or a global clock. Hence, all the communication and synchronization takes place through messages.

A distributed application consists of processes that communicate asynchronously with each other. These processes run on different nodes of the mobile system. The processes exchange information with each other through messages. For the application to run successfully, all the nodes on which the modules of the application are running should function properly. Node failures in the system are assumed to be fail-stop in nature. Henceforth, the term *node* will be used for both MHs and MSSs, unless explicitly stated otherwise.

The messages generated by the underlying distributed application will be referred to as the *computation messages*. Messages generated by the nodes to advance checkpoints, handle failures, and for recovery will be referred to as the *system messages*. Also, when a message of either type reaches a node, the node has the ability to peek at the message contents before actually processing it. Hence the reception/arrival of a message and its processing by the receiving node may not necessarily happen at the same time. They are two distinct events. The arrival of a message is recorded only on its processing.

3 Issues and Basic Ideas

Two major objectives in the design of a snapshot collection algorithm are efficiency and non-intrusiveness. A non-intrusive algorithm does not suspend the computation at the participating nodes during snapshot collection. Therefore, new inter-node dependencies may be created while global snapshot collection is in progress, which may lead to inconsistencies if not properly handled. An efficient algorithm forces a minimal set of nodes to take their local snapshots for each snapshot initiation, based on inter-node dependencies created since the last snapshot collection. As a result, the run-time overheads and the storage and communication overheads are kept low. A consequence of the efficiency and non-intrusiveness criteria is that the snapshot initiator does not know *a priori* the identity of all the nodes that will participate in the snapshot collection. This raises the issue of efficient termination detection of the snapshot collection process.

3.1 Issues

The mobility and energy consumption of the mobile hosts raise some issues not faced in a static distributed system.

3.1.1 Mobility

Changes in the location of an MH complicate the routing of messages. Messages sent by a node to another node may have to be rerouted because the destination node (MH) disconnected from the old MSS and is now connected to a new MSS. An MH may be disconnected from the network for a finite, but arbitrary period of time while switching from the old MSS to the new MSS. Routing protocols for the network layer, to handle node mobility, have been proposed in [2, 4, 11, 21, 25].

At the applications level, the checkpointing algorithm may generate a request for the disconnected MH to take its snapshot. Delaying a response to such a request, until the MH reconnects with some MSS, may significantly increase the completion time of the snapshot collection algorithm. So, an alternative solution is needed. One such solution is presented in Section 3.3.

There may be instances where the MH leaves the network, never to reconnect with it again. In such situations, it must be ensured that all the computations in which the MH is involved terminate before the MH quits the network. In the context of this paper, an efficient strategy to detect the termination of the snapshot collection algorithm is needed.

3.1.2 Energy Consumption

An MH is usually powered by a stand alone energy source, like a battery pack, that has to be replenished after a certain period of time. The mobility of an MH is directly dependent on its energy efficiency. The various components like the CPU, display, disk drive, etc. drain the battery. Message transmission and reception also consume energy. Energy consumption can be reduced by powering down individual components during periods of low activity [9]. This strategy is referred to as the *doze mode* operation [3].

Energy can be conserved, during snapshot collection, by forcing a minimal set of nodes to take their local snapshots. Otherwise, some MHs that have been dozing will be waken up by the snapshot collection. These MHs may not have participated in any computation for an extended period of time, and a new local snapshot of such MHs may not be required to create a consistent snapshot. Energy conservation and low bandwidth constraints are satisfied by reducing the number of system messages required to collect a consistent snapshot.

3.2 Minimal Dependency Information

Causality relationships are established through message communication. Node P_i maintains a boolean vector, R_i, of n components. At P_i, the vector is initialized as follows:

- $R_i[i] = 1$;
- $R_i[j] = 0$ if j≠i;

When node P_i sends a message to P_j, it appends R_i to the message. This informs P_j about the nodes that have causally affected P_i. While processing a message m, P_j extracts the boolean vector $m.R$ from the message and uses it to update R_j as follows: $R_j[k] \leftarrow R_j[k] \vee m.R[k]$, where $1 \leq k \leq n$. The processing of a message and the update of vector R_j take place as an atomic operation. This operation updates the dependency information. If the sender of a message is dependent on a node P_k before sending the message, the receiver will also be dependent on P_k on receiving the message. The spread of dependency information through messages is illustrated in Figure 1. P_4 is dependent on P_2 after receiving m_3. Since P_2 was dependent on P_1 before sending m_3, P_4 becomes (transitively) dependent on P_1 on receiving m_3.

Fidge [8] and Mattern [16] proposed vector clocks to maintain causality information. However, the overheads associated with vector clocks are high because a vector of n integers

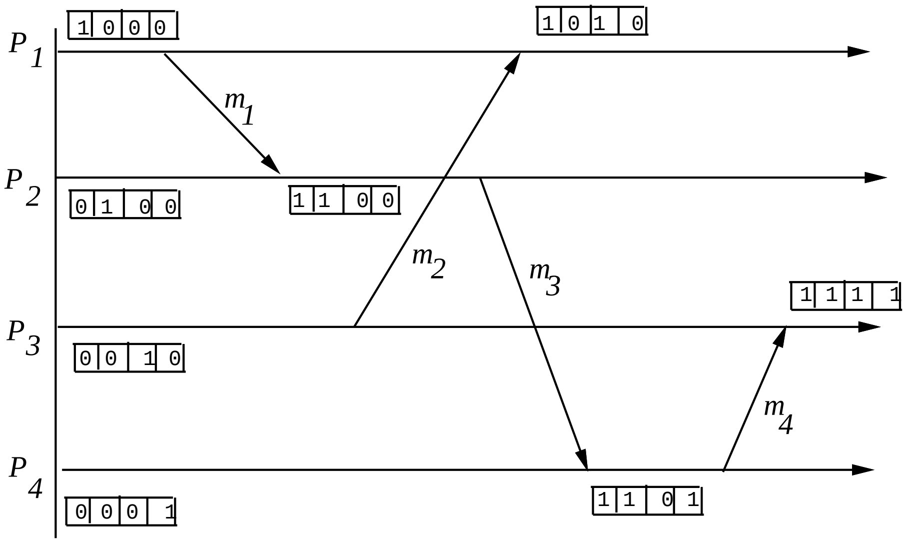

Figure 1: Propagation of dependency information.

is sent with each message. Assuming that each integer is stored in a 32-bit word, an n node system would have at least $4n$ bytes of overhead per message. As the word sizes of machines grow in the future, the overhead will also grow. This overhead can be quite debilitating for mobile systems because the links between $MH - MSS$ pairs are usually low bandwidth wireless links, and can severely affect the scalability of the mobile system. In comparison, the dependency vector only needs n bits of storage, and is independent of changes in the machine word size. Also, each update of the vector clock at a node, on receiving a message, requires up to n integer comparisons, as opposed to n bit-wise OR operations for the dependency vector. The bit operations are much faster than integer operations.

Use of the dependency information reduces the effort required to collect a global snapshot, as illustrated by the example in Figure 2. The vertical line S_1 represents the global snapshot at the beginning of the computation. Later, when P_2 initiates a new snapshot collection (at the instant marked by "X" on its time line), only P_3 and P_4 need to take their local snapshots because there are dependencies from these nodes onto P_2. Nodes P_1 and P_5 need not take their snapshots because they do not have dependencies onto P_2. The new global snapshot is represented by the cut S_2.

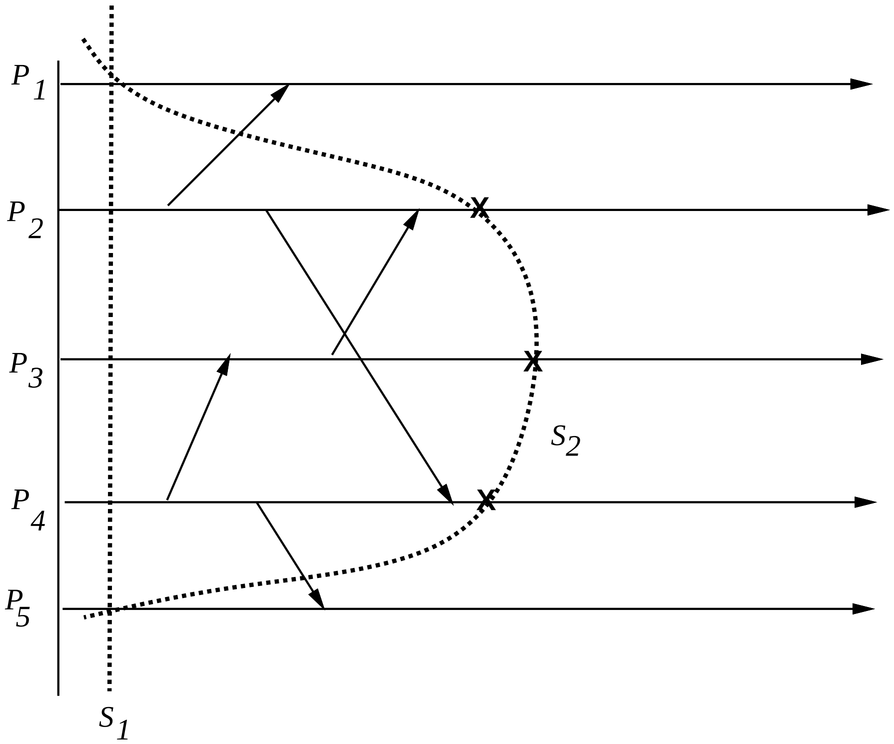

Figure 2: Local snapshots of minimal number of nodes taken.

3.3 Handling Node Mobility

Let a mobile host MH_i be initially connected to MSS_p. Then it disconnects from MSS_p and after a finite period of time connects with MSS_q. During the disconnect interval, only local events can take place at MH_i. Hence, no new dependencies, with respect to other nodes, are created during this interval. The dependency relation of MH_i, with the rest of the system, as reflected by its local snapshot, is the same no matter when the local snapshot is taken during the disconnect interval.

So, at the time of disconnecting from MSS_p, MH_i takes its local snapshot which is stored at MSS_p as $disconnect\_snapshot_i$. REQUESTs arriving at MSS_p to take MH_i's snapshot during the disconnect interval are serviced by using $disconnect\_snapshot_i$ as MH_i's local snapshot. The dependency vector of MH_i (R_i) at the time of taking the snapshot is used to propagate the snapshot request. Computation messages, meant for MH_i, arriving at MSS_p during the disconnect interval are buffered at MSS_p until the end of the interval.

The disconnect interval ends when MH_i connects to MSS_q and executes a *reconnect routine*. We assume that MH_i keeps in its stable storage the identity of the last MSS it was connected to (MSS_p). On connecting with MSS_q, the reconnect routine sends a query, through MSS_q, to MSS_p. If MH_i's stable storage does not contain the identity of its last MSS, for some reason, then the query is broadcast over the network. On receiving the query, MSS_p executes the following steps: if MSS_p had processed a snapshot request for MH_i during the disconnect interval, the corresponding snapshot ($disconnect\_snapshot_i$) and the buffered messages are sent to MH_i. If no snapshot request for MH_i was received by MSS_p during the disconnect interval, only the buffered messages are sent. Having sent the messages (and $disconnect\_snapshot_i$, if snapshot request was processed) MSS_p discards the buffered messages, $disconnect\_snapshot_i$ and the dependency vector of MH_i. When the data sent by MSS_p (buffered messages, and possibly $disconnect\_snapshot_i$) arrives at MH_i, MH_i executes the following action: all the buffered messages received from MSS_p are processed, and the dependency vector is modified to reflect the reception of these messages. If the received data contains $disconnect\_snapshot_i$, MH_i stores this snapshot as its local snapshot for the snapshot initiation, and resets all, except the i^{th} component, of the dependency vector, R_i, before processing the messages. Then the reconnect routine terminates, and the relocated mobile node MH_i can resume normal communication with other nodes in the system. As the old MSS discards the $disconnect\_snapshot_i$ at the end of the disconnect interval, an MH will not leave its local checkpoints at various MSSs in the fixed network.

Thus, the operations carried out by the MH and old MSS during the disconnect interval, and by the new and old MSSs and the MH during the reconnect routine hide the mobility of the MH from all the other nodes in the system. The FIFO communication of computation messages between pairs of nodes is preserved.

When an MH disconnects from an MSS, its state at the time of disconnection is available at the old MSS. So, instead of simply buffering the incoming messages for the MH, it is possible for the old MSS to process these messages on behalf of the disconnected MH. Variables, local to the MH, may be modified at the old MSS due to the reception of the messages. However, the local events at the disconnected MH may also modify the same variables. These modifications are being made to two different copies of the variables. It may be difficult to reconcile the inconsistencies that may arise due to these independent and concurrent modifications to the same variables. So this alternative is not very appealing.

In the reconnect routine, as described above, MSS_p sends $disconnect\_snapshot_i$ to MH_i if MSS_p has processed a snapshot request for MH_i during the disconnect interval. Alternatively, MSS_p can ask the relocated MH_i to take a new local snapshot before processing the buffered messages. The consistency of a global snapshot remains unaffected as $disconnect\_snapshot_i$ and the new local snapshot of MH_i reflect the same dependency relation, with respect to the rest of the system, and can be substituted for each other. Moreover, precious bandwidth, that would have been used to send $disconnect\_snapshot_i$ during the reconnect routine, is saved. However, this alternative is not suitable under all circumstances. Let us assume that the global snapshot is being collected to evaluate the value of a predicate. During the snapshot collection, $disconnect\_snapshot_i$ is used as MH_i's local snapshot, and the predicate returns a certain value. The local events at MH_i, during the disconnect interval, may modify some local variables which will be reflected in the new local snapshot of MH_i taken during the reconnect routine. Later, if the predicate is evaluated for the same global snapshot, after MH_i has connected to MSS_q, it may return a different value. The same system state may appear to be returning different values for a predicate. The alternative described here can be employed if the local events at MH_i, during the disconnect interval, do not modify the variables on which the value of the predicate is dependent.

Section 4 presents an algorithm to collect the snapshot of a mobile distributed system. It addresses the issues raised in Section 3.1, and is more efficient than the snapshot collection algorithms for static distributed systems proposed in the past. In the algorithm, no distinction is made between mobile and static nodes as the mobility of the nodes can be hidden, as described above.

4 Minimal Snapshot Collection Algorithm

In this section, we present a non-blocking snapshot collection algorithm for mobile computing systems. The algorithm forces a minimal set of nodes to take their local snapshots. Thus the effort required for snapshot collection is reduced, and nodes that have been dozing are unlikely to be disturbed. Moreover, the algorithm is non-intrusive.

After the coordinated snapshot collection terminates, the nodes that did not participate in the snapshot collection take their local snapshots in a quasi-asynchronous fashion. This reduces the amount of computation that has to be undone on node failures. Huang's algorithm [10] is employed to detect the termination of the coordinated snapshot collection. Unlike [5, 12], information about termination is not propagated along a tree rooted at the snapshot initiator. Instead, the nodes send this information directly to the initiator. Hence, termination detection is fast and inexpensive.

We assume that at any time, at most one snapshot collection is in progress. Techniques to handle concurrent initiations of snapshot collection by multiple nodes can be found in [18].

4.1 Data Structures

Each node maintains the following data structures:

interval_number : an integer value maintained at each node that is incremented each time the node takes its local snapshot.

interval_vector : an array of n integers at each node, where interval_vector[j] indicates the interval_number of the next message expected from node P_j. For node P_i, interval_vector[i] is equal to its interval_number.

trigger : a tuple (*pid*, *inum*) maintained by each node. *pid* indicates the snapshot initiator that triggered this node to take its latest checkpoint. *inum* indicates the interval_number at node *pid* when it took its own local snapshot on initiating the snapshot collection. *trigger* is appended to every system message and the first computation message that a node sends to every other node after taking a local snapshot.

send_infect : a boolean vector of size n, maintained by each node in its stable storage. The vector is initialized to all zeroes each time a snapshot at that node is taken. When a node P_i sends a computation message to node P_j, it sets send_infect[j] to 1. Thus this vector indicates the nodes to which computation messages have been sent since the last checkpoint, or since the beginning of the computation, whichever is later.

propagate : a boolean vector of size n, maintained by each node in its stable storage. It is used to keep track of the nodes to which snapshot REQUESTs were sent by the node. The vector is initialized to all 0's.

weight : a non-negative variable of type *real*, with maximum value of 1. It is used to detect the termination of the snapshot collection.

The interval_numbers and interval_vectors are initialized to 1 and an array of 1's, respectively, at all the nodes. The trigger tuple at node P_i is initialized to $(i, 1)$. The weight at a node is initialized to 0.

When node P_i sends any message, it appends its interval_number and the dependency vector, R_i, to the message.

4.2 The Algorithm

The algorithm does not require any node to suspend its underlying computation. When P_i initiates a snapshot collection, it takes a tentative local snapshot, increments its interval_number, sets *weight* to 1, and stores its own identifier and the new interval_number in trigger. It then sends snapshot REQUESTs to all the nodes P_j, such that $R_i[j] = 1$ and resumes its computation. Each REQUEST message carries the trigger of the initiating node, the vector R_i and a portion of the weight. The weight of the REQUEST sender is decreased by an equal amount. Also, when a node REQUESTs another node to take its snapshot on behalf of the initiator, it appends the initiator's trigger tuple and a portion of the received weight to all those REQUESTs.

When a snapshot REQUEST is received by a node P_i and *request.trigger* is not equal to P_i*.trigger*, P_i takes a tentative local snapshot and sends REQUESTs to all the nodes that have their corresponding bits set in its dependency vector, R_i, but not in the vector $m.R$ carried by the received REQUEST. P_i then resumes its underlying computation. As already explained in Section 3.3, if P_i is an MH and the REQUESTs are generated during its disconnect interval, then the operations described above are carried out on its behalf by the MSS to which it was previously connected.

If *request.trigger* is equal to P_i*.trigger* when P_i receives the REQUEST, it means that P_i has already taken its snapshot for this snapshot initiation. So, P_i does not take a local snapshot. But,

- if the *propagate* vector has no 1's in it, then a RESPONSE is sent to the snapshot initiator with a weight equal to the weight received in the REQUEST.
- if *propagate* has some bits set to 1, then for all j such that $propagate[j] = 1$, a REQUEST is sent to P_j with a non-zero portion of the weight received in the REQUEST. Then the *propagate* vector is reset to all 0's and the remaining portion of the received weight is sent to the initiator in a RESPONSE.

The bits in *propagate* are set when REQUESTs are sent to nodes on the reception of computation messages, as described later in this Section. Note that the trigger, carried by the REQUEST messages, prevents a node from taking multiple snapshots when the node receives multiple REQUESTs for the same global snapshot initiation.

Since the computation at any node does not block after it has taken a snapshot, the following scenario is possible: A node P_j takes its snapshot and then sends a computation message m to node P_k. Node P_k receives (and processes) this message before it receives a REQUEST messages to take its snapshot. This will lead to an inconsistency in the global snapshot – the snapshot taken by P_k will be causally dependent upon the snapshot of P_j. This problem is solved by having a node include its trigger in the first computation message it sends to every other node after taking its snapshot. P_j checks if $send\_infect[k] = 0$ before sending a computation message to P_k. If so, then it sets $send\_infect[k]$ to 1 and appends its trigger to the message. When P_k receives this message from P_j, by looking at the trigger in the message, P_k can infer that P_j has taken a new snapshot before sending the message. Consequently, P_k takes its tentative snapshot before processing the message. Later, if P_k receives a REQUEST message for the snapshot initiation, it knows that a local snapshot has already been taken (the local trigger is equal to the trigger in the message).

When P_j receives a computation message m from P_i, it compares the interval_number received in the message with its own interval_vector$[i]$. If the interval_number received is less than or equal to interval_vector$[i]$, then the message is processed and no snapshot is taken.

If the interval_number of the computation message received is greater than interval_vector$[i]$, then it implies that P_i has taken a snapshot before sending the message, and this message is the first computation message sent by P_i to P_j since P_i's snapshot. So, the message must have the trigger tuple. The following steps are executed in such a situation:

1. P_j's interval_vector$[i]$ is updated to the interval_number in the message received from P_i.
2. P_j checks the trigger tuple of the message received. For the sake of convenience, we shall call the trigger tuple at P_j as *own_trigger* while the trigger tuple received in the message as *msg_trigger*.
   (a) if $msg\_trigger = own\_trigger$, it means that the latest snapshots of P_i and P_j were both taken in response to the same snapshot initiation event. So no action needs to be taken besides updating the dependency vector, R_j.
   (b) if $msg\_trigger.pid = own\_trigger.pid \wedge msg\_trigger.inum > own\_trigger.inum$, it means that P_i has sent the message after taking a new snapshot, while P_j has not taken a snapshot for this snapshot initiation. So P_j takes a tentative snapshot before processing the message and the tuple *own_trigger* is set to be equal to *msg_trigger*. P_j also propagates the snapshot request by sending REQUESTs to all the nodes that have their corresponding bits set in R_j, but not in the bit-vector $m.R$ of the message received. For every such REQUEST message sent out to node P_k, $propagate[k]$ is set to 1.
   (c) if $msg\_trigger.pid \neq own\_trigger.pid$, there are two possibilities:
      i. if P_j has not processed any message satisfying the condition $msg\_trigger.pid \neq own\_trigger.pid$ since its last local snapshot, then P_j takes its tentative snapshot and sets *own_trigger* to *msg_trigger* before processing the message. Then P_j propagates the snapshot REQUEST, using the dependency vector R_i, as described earlier.
      ii. if P_j has already processed a message from any node, satisfying the condition $msg\_trigger.pid \neq own\_trigger.pid$ since its last local snapshot, then no new

local snapshot needs to be taken.

The snapshot initiator adds the weights received in the RESPONSEs to its own weight. When its weight becomes equal to 1, it concludes that all the nodes involved in the snapshot collection have taken their tentative local snapshots and sends out COMMIT messages to all the nodes from which it received RESPONSEs. The nodes turn their tentative snapshots into permanent ones on receiving the COMMIT message.

The length of the longest path that the initiator's REQUEST has to propagate through is limited by the number of nodes in the system. Hence, REQUEST propagation takes finite time. Once a REQUEST arrives, local snapshot is taken, and RESPONSE is sent in finite time. Thus, all the RESPONSEs are sent to the initiator within a finite period of time of the snapshot initiation. The initiator sends the COMMIT after getting all the RESPONSEs, at which stage its weight becomes equal to one. Thus the snapshot collection process terminates in a finite amount of time, provided there are no node or link failures during the snapshot collection process.

The pseudo-code for this algorithm is presented in Figure 3.

4.3 An Example

The operation of the algorithm can be better understood with the aid of the example presented in Figure 4. Node P_2 initiates the snapshot collection by taking its local snapshot at the instant marked by "X". There are dependencies from P_1 and P_3 to P_2. So, REQUEST messages (indicated by broken arrows) are sent to P_1 and P_3 to take their snapshots. P_3 sends a message m_4 to P_1 after taking its snapshot. When m_4 reaches P_1, it is the first message received by P_1 such that $msg\_trigger.pid \neq own\_trigger.pid$. So, P_1 takes its snapshot just before processing m_4.

Node P_0 that has not yet communicated with any other node, takes a local snapshot independent of any other snapshot collection process. Later, it sends a message m_5 to P_1. As a result of P_0 taking an independent local snapshot, the interval number of m_5 is higher than the value expected by P_1 from P_0. But when m_5 reaches P_1, it is not the first

type trigger = record (pid : node_id; inum : integer;) **end**
var own_trigger, msg_trigger : trigger;
interval_vector : array[1..n] of integer;
interval_number, rfirst : integer; weight: real;
R_i, Propagate, Temp2, vector, first : bit vector of size n;

Actions taken when P_i sends computation message to P_j

if first[j]=0 *then* {first[j]$\leftarrow$1; send(P_i, message, R_i, interval_number, own_trigger); }
else send(P_i, message, R_i, interval_number, NULL);

Action for snapshot initiation by P_j

clear first; rfirst$\leftarrow$0; take local snapshot; weight$\leftarrow$1.0
own_trigger.pid$\leftarrow$own_identifier(P_j); increment(interval_number);
own_trigger.inum$\leftarrow$interval_number; increment(interval_vector[j]);
to all nodes P_i,such that R[i]=1
{ weight$\leftarrow$weight/2; send_weight$\leftarrow$weight;
send(initiator_id, REQUEST, jr, interval_number, own_trigger, send_weight);}
reset all bits, except own bit, in the dependency vector R_j;
resume normal computation;

Other nodes, P_i, on receiving snapshot request from P_j

receive(P_j, REQUEST, $m.R$, *interval_number'*, msg_trigger, recv_weight);
if msg_trigger = own_trigger *then*{
to all nodes P_k,such that Propagate[k]=1
{ recv_weight$\leftarrow$recv_weight/2; send_weight $\leftarrow$recv_weight;
send(P_i, REQUEST, R_i, interval_number, own_trigger, send_weight);}
send(P_i, RESPONSE, recv_weight) to initiator;}
else { interval_vector[j]$\leftarrow$ *interval_number'*;
propagate_snapshot(R_i, $m.R$, P_i, interval_number, msg_trigger, recv_weight);
Propagate$\leftarrow$all 0's;
resume normal computation;}

Action for node P_i, on receiving computation message from P_j

receive(P_j, REQUEST, $m.R$, *interval_number'*, msg_trigger);
if *interval_number'* $\leq$ interval_vector[j] *then* process the message and exit;
else { interval_vector[j]$\leftarrow$ *interval_number'*;
if msg_trigger.pid = own_trigger.pid *then*
{ *if* msg_trigger.inum = own_trigger.inum *then* process the message;
else{ propagate_snapshot(R_i, $m.R$, P_i, interval_number, msg_trigger,0);
process the message; rfirst$\leftarrow$1;} }
else{ *if* rfirst = 0 *then*
{ propagate_snapshot(R_i, $m.R$, P_i, interval_number, msg_trigger,0);
process the message; rfirst$\leftarrow$1;}
else process the message;} }

propagate_snapshot(R_i, $m.R$, P_i, interval_number, msg_trigger, recv_weight)

{take local snapshot;
increment(interval_number); increment(interval_vector[i]);
own_trigger$\leftarrow$msg_trigger;
Propagate$\leftarrow R_i$ - $m.R$; Temp2$\leftarrow R_i$ OR $m.R$;
to all nodes P_k, such that Propagate[k]=1{
recv_weight$\leftarrow$recv_weight/2; send_weight $\leftarrow$recv_weight;
send(P_i, REQUEST, Temp2, interval_number, own_trigger, send_weight);}
reset all bits, except own bit, in the dependency vector R_i;
rfirst$\leftarrow$0;}

Figure 3: Non-blocking snapshot collection algorithm

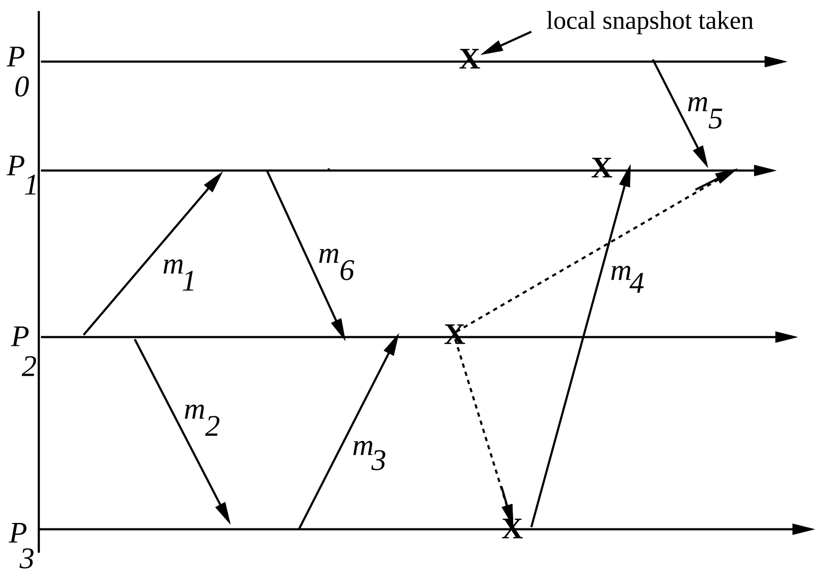

Figure 4: An example snapshot collection.

computation message received by P_1 with a higher interval number than expected whose *msg_trigger.pid* is different from P_1's *own_trigger.pid* since the last snapshot. So a snapshot is not taken, as explained in step 2(c)ii (because it will lead to inconsistency — the reception of m_4 will be recorded if P_1 takes a snapshot just before it processes m_5, but the transmission of m_4 will not have been recorded by P_3). Yet another reason for not taking a new snapshot each time a computation message with a higher interval number than expected is received is that it may lead to an *avalanche effect*. For example, in Figure 4, if a snapshot is taken before processing m_5, then P_3 will have to take another snapshot to maintain consistency. If, in the meanwhile, P_3 has received a message since it sent m_4, then the sender of that message has to take a snapshot. This chain may never end!

The snapshot REQUEST sent by P_2, to P_1, reaches P_1 after P_1 has taken a local snapshot on the arrival of the computation message m_4. So, *msg_trigger* (of the REQUEST) is equal to *own_trigger*. Hence, the snapshot REQUEST is ignored, as explained in part 2a of the algorithm.

4.4 Discussion

In the algorithm, only the nodes on which the initiator is dependent are *forced* to take their snapshots. During the coordinated snapshot collection, nodes are made to take their local snapshots on the arrival of REQUEST messages, or computation messages with higher than expected interval numbers. These snapshots are committed on the termination of the coordinated snapshot collection phase. This is called *aggressive* checkpoint advancement. Once the coordinated snapshot collection terminates, other nodes, on which the initiator is not dependent (either directly or transitively) advance their checkpoints in a *lazy* fashion when they receive the first computation message with a higher than expected interval number.

For example, let the system shown in Figure 4 have an additional node P_4. Let P_3 send a computation message to P_4 before taking its local snapshot in response to P_2's request. If P_4 has not been involved in any other communication, it will not take a snapshot for the snapshot collection initiated by P_2 (no aggressive checkpoint advancement at P_4). But, if P_3 sends yet another computation message to P_4 after taking its local snapshot, then P_4 will take its local snapshot before processing this message (advancing the checkpoint in a lazy manner). The checkpointing by nodes, during the lazy phase, even though driven by message receptions, simulates a quasi-asynchronous checkpointing. So, it may not be necessary to initiate synchronous checkpointing frequently.

Lazy checkpoint advancement is especially suited for mobile computing systems. A steady advancement of the checkpoint during the lazy phase precludes the need for frequent initiations of coordinated snapshot collection. Infrequent initiations of snapshot collection cause the imposition of the high checkpointing overheads of coordinated snapshot collection on the low bandwidth network only occasionally. Besides, the lazy advancement is due to transmission of computation messages. So, it imposes little overheads of its own. It also prevents the global snapshot from getting out of date. So, the amount of computation that may have to be undone during recovery from node failures is minimized.

Thus, the snapshot collection algorithm is a combination of aggressive and lazy advancements of checkpoints. If a node fails during the aggressive phase, all the tentative snapshots are discarded and the snapshot collection has to be initiated all over again.

4.5 Proof of Correctness

Lemma 1 *: If node P_i takes a snapshot $\wedge$ $R_i[j]=1$, then P_j takes a snapshot for the same snapshot initiation.*

Proof: If node P_i initiates snapshot collection, it sends REQUESTs to all P_j such that R_i[j]=1. If P_i is not the snapshot initiator and takes its snapshot on receiving a REQUEST from P_k, then for every node P_j such that $R_i[j]$=1, there are two possibilities:

Case 1: If $m.R[j]$=0 in the REQUEST received by P_i from P_k, then P_i sends a REQUEST to P_j.

Case 2: If $m.R[j]$=1 in the REQUEST received by P_i from P_k, then a REQUEST has been sent to P_j by at least one node in the snapshot REQUEST propagation path from the snapshot initiator to P_k.

So, at least one snapshot REQUEST is sent to P_j. If P_j is a static host, then the underlying network will route the REQUEST to it. IfP_j is an MH, and P_i's knowledge of P_j's location indicates that the latter is connected to MSS_p, then there are three distinct possibilities when the REQUEST reaches MSS_p:

1. P_j is still connected to MSS_p: the REQUEST is forwarded to P_j.
2. P_j is disconnected from the network: MSS_p takes a snapshot on behalf of P_j by converting $disconnect\_snapshot_j$ into a tentative local snapshot for P_j.
3. P_j has reconnected to MSS_q: MSS_p forwards the REQUEST to MSS_q, as explained in Section 3.3.

Thus, if a node takes a snapshot, every node on which it is directly dependent receives at least one snapshot REQUEST.

There are two possibilities when P_j receives the first snapshot REQUEST:

1. P_j has not taken its snapshot when the first snapshot REQUEST for this initiation arrives: P_j takes its snapshot on receiving the REQUEST message.

2. P_j has taken a snapshot for this snapshot initiation when the first snapshot REQUEST arrives: this REQUEST and all subsequent REQUESTs for this initiation are ignored. (The snapshot was taken when the first computation message with a higher than expected interval_number is received since the node's last snapshot. The msg_trigger carries the identity of the snapshot initiator.)

Hence, when a node takes a snapshot, every node on which it is directly dependent takes a snapshot. ∎

Applying the transitivity property of the dependence relation, every node on which the initiator is dependent, directly or transitively, takes a snapshot. These dependencies may have been present before the snapshot collection was initiated, or may have been created while the coordinated snapshot collection (aggressive phase) was in progress.

Theorem 1 *: The algorithm ensures consistent global snapshot collection.*

Proof: In order to prove the theorem, we have to prove that: *If the reception of a message has been recorded in the snapshot of a node, then the corresponding transmission has been recorded in the snapshot of some other node.*

Let P_i record the reception of message m from P_j in its snapshot. So, $R_i[j]$=1 at P_i at the time of taking its snapshot. From Lemma 1, P_j's snapshot, too, is taken. There are three possible situations under which P_j's snapshot is taken:

Case 1: P_j's snapshot is taken due to a REQUEST from P_i. Then:

send(m) at $P_j \rightarrow$ receive(m) at P_i

receive(m) at $P_i \rightarrow$ snapshot taken at P_i

snapshot taken at $P_i \rightarrow$ REQUEST sent by P_i to P_j

REQUEST sent by P_i to $P_j \rightarrow$ snapshot taken at P_j

Using the transitivity property of $\rightarrow$, we have: send(m) at $P_j \rightarrow$ snapshot taken at P_j. Thus sending of m is recorded at P_j.

Case 2: P_j's snapshot is taken due to a REQUEST from a node P_k, $k \neq i$. Let us assume that P_j sends m after taking its local snapshot implying that when m arrives at P_i, its interval_number is greater than interval_vector[j] at P_i. So, P_i takes its snapshot before processing m. Hence, reception of m is not recorded in the snapshot of P_i — a contradiction of the starting assumption that P_i had recorded the reception of the message. So, P_j must have sent m before taking its local snapshot.

Case 3: P_j's snapshot is taken due to the arrival of a computation message m' at P_j from P_k. Let us assume that m' has been received and local snapshot has been taken at P_j before P_j sends m to P_i. This is similar to Case 2, and leads to a similar contradiction. So, m' must have been received after sending m and the transmission of m must have been recorded in P_j's snapshot.

Thus, if the reception of a message is recorded in the snapshot, then its transmission must have been recorded in the snapshot. ∎

When snapshots are taken on the arrival of computation messages, the higher than expected interval_number in the message has the same effect as a piggybacked snapshot REQUEST. Piggybacking control information, to distinguish between messages sent by a node before and after its snapshot, is a strategy used for consistent snapshot collection in systems where communication channels are non-FIFO and computation messages sent after taking a snapshot may overtake the snapshot collection marker [13]. Therefore, the proposed algorithm can be easily modified for consistent snapshot collection of systems where message communication is non-FIFO.

5 Recovery from a Failure

To recover from node failures, the system should be restored to a consistent state before the computation can proceed. Failure recovery algorithms for static distributed systems have not considered the issues pertinent to mobile networks. In most of these recovery algorithms, if a node fails and has to roll-back to its local checkpoint, *all* the other nodes are also rolled-back to their checkpoints. This is an expensive recovery method for two reasons: Firstly, it may involve unnecessary node roll-backs. If the computation at a node P_i is not dependent on an operation that was undone due to the roll-back of the faulty node, then P_i should not be made to roll-back. Furthermore, MHs that are *dozing* may be waken up to carry out roll-backs that are not essential for maintaining consistency. Secondly, several nodes in the system are mobile. If all the nodes have to roll-back their computations, several messages will be required at the network layer to locate all the mobile nodes. By keeping the number of nodes that need to roll back to a minimum, message traffic (for locating nodes) can be reduced, thus conserving the limited bandwidth.

In some recovery algorithms for static distributed systems, no non-faulty node is made to roll-back its computation [7, 23]. However, [23] requires extensive logging of message contents, both at the sender and receiver ends. In [7] the antecedence graph, containing the entire causal relationship, is kept in volatile storage and periodically copied to stable storage. Each computation message has to carry portions of the antecedence graph, significantly increasing the size of the messages. The MHs have a limited memory. So, extensive logging of information is not feasible. Having the MSSs maintain the logs on behalf of the MHs will lead to movement of large amounts of data over the static network as an MH moves from one MSS to another. The bandwidth of the channel between an MH and an MSS being low, supporting high overhead computation messages of the type described above will be difficult.

We propose a recovery algorithm for mobile computing systems where, instead of all the nodes, only a minimal set of nodes is made to roll back the computation, and extensive logging of messages on stable storage is not required. The concept of dependency is used to minimize the number of nodes that roll back their computations. For example, suppose a node P_i fails and has to roll back. If no new dependency from P_i to P_j has been created since P_j's last checkpoint, there is no need for P_j to roll-back in response to P_i's roll-back. Only those nodes that have a dependency on the failed node since the latter's last checkpoint need to roll back, to maintain global consistency.

5.1 Roll-back to a Consistent State

Each node keeps track of all the nodes to which it has sent computation messages, using the *send_infect* vector. To recover from a failure, node P_i rolls-back to its latest checkpoint and sends roll-back requests to all the nodes whose bits are set in the *send_infect* vector. The *send_infect* vector is sent with the roll-back requests. When a node P_j receives the first roll-back request, it takes the following actions:

1. P_j rolls-back to its latest checkpoint.
2. P_j sends a roll-back request to every node P_k whose bit is set in P_j's send_infect vector but is 0 in the bit-vector received in the roll-back request message. The vector obtained by bit-wise ORing of P_j's send_infect vector and the received bit-vector is sent with each request.

All subsequent roll-back requests received by P_j, originating due to this failure of P_i, are ignored. A data structure similar to trigger (Section 4.1) can be used to indicate the node that initiated the roll-back.

The node that initiates roll-back has an initial weight of one. As in Huang's termination detection algorithm [10], a portion of this weight is sent by the initiator with each roll-back request. Each time a node propagates the roll-back request, it sends a portion of its weight with the corresponding messages. It also sends its residual weight back to the initiator on rolling-back. The roll-back phase terminates when the initiator's weight becomes equal to one. At the end of this phase, the system has been restored to a consistent state. The roll-back requests for MHs are rerouted to them by the MSS to which they were previously connected, through the MSS to which they are currently connected. The strategies proposed in [2, 4, 11, 21, 25] can be employed to locate the new MSS to which the MH is connected.

5.2 Retracing the Lost Computation

Once the system has rolled back to a consistent state, the nodes have to retrace their computation that was undone during the roll-back. The following kinds of messages have to be handled while retracing the lost computation:

- *Orphan messages*: Messages whose reception has been recorded, but the record of their transmission has been lost. This situation arises when the sender node rolls-back to a point prior to sending the messages while the receiver node still has the record of its reception.

- *Lost messages*: Messages whose transmission has been recorded, but the record of their reception has been lost. This happens if the receiver rolls-back to a point prior to the reception of the messages, while the sender does not roll-back to a point prior to their sending.

- *Out of sequence messages*: This situation arises when the messages do not arrive at the recovering node in the same order as they did originally. For example, let P_i send two messages m_1 and m_2 to P_j. Node P_j rolls-back after receiving m_1, and at that time let m_2 be in transit from P_i to P_j. When P_j requests P_i to resend the lost messages, m_1 is sent once again. The communication links being FIFO, the second copy of m_1 reaches P_j after m_2.

- *Duplicate messages*: This happens when more than one copy of the same message arrives at a node; perhaps one corresponding to the original computation and one generated during the recovery phase. The first copy has been processed. So, the subsequent copies should be discarded. A transparent recovery algorithm should never generate duplicate output messages to the environment.

The proposed recovery algorithm maintains data structures similar to those in [23]; however, it logs messages in volatile storage only at the sender.

1. Whenever a node sends a computation message, it maintains a copy of it in volatile storage until the checkpoint at the node (determined by the snapshot collection algorithm) is advanced to a point in the future of the message transmission event.

2. Each node maintains two integer vectors in stable storage: sent$[1..n]$ and received$[1..n]$ where n is the number of nodes. At node P_i, sent$[j]$ and received$[j]$ are equal to the number of computation messages sent to and received from node P_j, respectively, since the beginning of the computation. Both the vectors are initialized to zeroes.

3. Each node logs the order in which messages have been received from all the other nodes, since its last checkpoint, in a QUEUE maintained in stable storage.

Input messages, received by a node from the external environment, are logged in stable storage before being processed.

The logs and data structures mentioned above are used for recovery in the following manner: During normal operation, whenever P_i sends a computation message to P_j, it increments sent$[j]$ by one and stamps the outgoing message with the new value of sent$[j]$. Whenever P_i receives a computation message from P_j, it increments received$[j]$ by one. P_i also adds an entry for P_j to the tail of the QUEUE. When P_i rolls back to its latest checkpoint, it resets received$[1..n]$ to the values corresponding to the state at the checkpoint. The node also sets a pointer to point to the location of the QUEUE corresponding to the entry immediately following the checkpoint. If no message reception was undone during the roll-back, there is no such entry and the pointer is set to null.

1. Having rolled back, when node P_i starts recovery, it broadcasts a RECOVERING(i) message that contains the vector received$[1..n]$.
2. When a node P_j receives the RECOVERING(i) message, it retransmits copies of the messages meant for P_i, in its volatile storage, whose sent$[i]$ values are greater than the received$[j]$ value in the broadcast message.
3. After broadcasting the RECOVERING(i) message, the incoming messages, m, from the other nodes, P_j for all $j \in \{1, \ldots, n\}$, are received and processed in the following manner:

   (a) If sent$[i]$ in the message is equal to received$[j]$+1 at P_i, and the pointer is non-null and pointing to P_i in the QUEUE, then the message can be immediately processed. The pointer is moved to the next entry in the QUEUE.

   (b) If the sent$[i]$ value in the message is less than or equal to received$[j]$, then it is a duplicate message and is ignored.

   (c) Otherwise, the message has been received out of sequence – there are messages from other nodes to P_i that have to be processed first. So, m is buffered and not processed until the condition specified in 3a is satisfied.

Orphan messages cannot arise during roll-back and recovery because whenever P_i rolls-back after sending a message to P_j, it also sends a rollback request to P_j since send_infect$[j]$ has been set to 1. Hence, P_j rolls-back erasing the record of the reception of the message. The problem of *lost messages* is solved by logging messages in volatile storage at the sender. These messages can be retransmitted, during recovery, on receiving the RECOVERING message. *Out of sequence* messages and *duplicate* messages can be handled as mentioned above in 3b and 3c, respectively. In order to prevent duplicate output messages from being sent, an output message is not sent to the external environment until the checkpoint is advanced to a point that is in the future of the generation of the output message.

6 Comparison With Earlier Work

In the Chandy-Lamport algorithm [6], which is one of the earliest snapshot collection algorithms for a system with static nodes, system messages are sent along all the links in the network during snapshot collection. This leads to a message complexity of $O(n^2)$. In the proposed snapshot collection algorithm, system messages need not be sent along all the links in the network. The number of system messages required is proportional to the number of channels in the interconnection network along which computation messages have been sent since the last snapshot collection. Therefore, the average message complexity of the proposed algorithm is lower than Chandy-Lamport's algorithm.

Acharya *et al.* [1] were the first to present an asynchronous snapshot collection algorithm for distributed applications on mobile computing systems. They give two reasons why they consider synchronous checkpointing to be unsuitable for mobile systems: (i) high cost of locating MHs because in the Chandy-Lamport kind of algorithm, an MH has to receive REQUESTs along every incoming link and (ii) non-availability of the local snapshot of a disconnected MH during synchronous checkpointing. The synchronous algorithm proposed in this paper overcomes both these shortcomings: by conveying the transitive closure of dependency information through R_i, the number of REQUESTs is reduced, thus reducing the cost of locating the MHs. Also, the local snapshot of a disconnected mobile host MH_i is always available, as $disconnect\_snapshot_i$, at the MSS to which it was last connected.

In [1], an MH has to take its snapshot whenever a message reception is preceded by a message transmission at that node. This may lead to as many local snapshots being taken as the number of computation messages (if the transmission and reception of messages are interleaved). This is likely to impose a high checkpointing cost on the nodes. Considering that message communication is much more frequent than initiations of synchronous snapshot collection, or movement of MHs from one MSS to another, the proposed algorithm will require the nodes to take their local snapshots much less frequently than the algorithm in [1]. The lazy checkpoint advancement in the proposed algorithm overcomes yet another potential drawback of synchronous checkpointing. During the lazy phase, messages needed for checkpoint advancement are spread over a period of time, rather than being bursty during a short duration. Such low density traffic is suitable for the low bandwidth communication networks of the mobile computing systems.

In Venkatesan's algorithm [22], a node sends out *markers* (corresponding to REQUESTs in the proposed algorithm) on all the outgoing edges along which computation messages have been sent since the last checkpoint. However, as already explained in Section 3, in order to efficiently collect a consistent snapshot, checkpointing REQUESTs need only be propagated from the receiver of messages to the sender, not the other way round as in [22]. Therefore, the proposed algorithm, because it propagates checkpointing decision in the receiver to sender direction, makes a minimal set of nodes to take its snapshot and is more suited for mobile computing systems than the algorithm given in [22].

The main advantage of our algorithm over the synchronous Koo-Toueg algorithm [12] is that the underlying computation is never suspended during snapshot collection by our algorithm. This significantly reduces the run-time overheads of the algorithm. Moreover, in the snapshot collection algorithm of Koo-Toueg, only direct dependencies are maintained, as opposed to transitive dependencies maintained by our algorithm. Snapshot requests propagate faster along the transitive dependency chain as compared to the direct dependency chains. Knowledge about transitive dependencies also reduces the number of snapshot REQUEST messages required by the algorithm. In [12] a node P_i sends snapshot requests to all the nodes P_j on which it is *directly* dependent. In our algorithm, if P_i knows that a REQUEST has already been sent to P_j by some other node, then it does not send a REQUEST to P_j. This information is carried by the bit-vector of the REQUEST messages.

In Venkatesan-Juang's optimistic failure recovery algorithm [24], no dependency information is sent with the computation messages. However, several iterations may be needed for all the nodes to roll-back to mutually consistent states, at the time of recovery. This is a high price to pay, considering that the low overheads associated with the computation message of our algorithm help accomplish consistent roll-back in one iteration. Moreover, the mobile nodes may change their location between iterations, complicating the roll-back process.

The recovery algorithms proposed in [7, 23] ensure that only the faulty nodes are rolled back. However, they require extensive logging of messages and high communication overheads. The algorithm proposed in this paper has lower storage and communication overheads than [7, 23] and may require few non-faulty nodes to roll-back. Our algorithm has slightly higher communication overheads than Venkatesan's algorithm [22], but much smaller delays. Thus our algorithm has slightly higher overheads than the most economical recovery algorithm, and a slightly greater delay than the fastest recovery algorithm, however it does not suffer from the drawbacks, i.e., extensive logging and high communication overheads, of either. So, it is ideal for mobile computing systems where storage and communication bandwidth are at a premium, and time constraints are not very rigid.

7 Conclusions

A mobile computing system consists of mobile and stationary nodes, connected to each other by a communication network. The availability of such systems is on the rise due to the proliferation of portable computers and advances in communication technology. An efficient recovery mechanism for mobile computing systems is required to maintain continuity of computation in the event of node failures. In this paper, we have developed low overhead snapshot collection and recovery algorithms for distributed applications in a mobile computing system, that meet the requirements of node mobility, energy conservation, and low communication bandwidth.

Dependency information among nodes is used to incrementally advance the global snapshot of the system in a coordinated manner and to determine the minimal number of nodes that need to roll-back in response to node failures. The proposed snapshot collection algorithm is non-intrusive — does not require the participating nodes to suspend their computation during snapshot collection. The lazy advancement of the checkpoint, after coordinated snapshot collection terminates, leads to the checkpointing overheads being amortized over a period of time. As the underlying computation is never suspended during snapshot collection, the run-time overheads are low. Each system message has a small size, and incurs a low overhead as the information about dependencies can be conveyed using just a bit-vector. This compares favorably with existing implementations like Manetho [7] where the antecedence graph, incorporating information about the exact ordering of message transmissions and receptions at the nodes, is piggybacked on each message.

We used dependency information to develop a minimal recovery algorithm. Consequently, the computation that is lost due to roll-backs is less than that in a number of algorithms proposed for static distributed systems. The recovery algorithm has low storage and communication overheads. The time to recover from node failures and to restore the system to a consistent state is less than that needed by some of the most economical (low overheads) recovery algorithms. Our recovery algorithm is a compromise between two diverse recovery strategies — fast recovery with high communication and storage overheads, and slow recovery with very little communication overheads. Hence, the algorithm is suitable for mobile computing systems.

In summary, we have provided efficient techniques that are suitable for snapshot collection and recovery in mobile computing systems.

References

[1] A. Acharya, B. R. Badrinath, and T. Imielinski. Checkpointing Distributed Applications

on Mobile Computers. Technical report, Department of Computer Science, Rutgers University, 1994.

[2] B. Awerbuch and D. Peleg. Concurrent Online Tracking of Mobile Users. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, 1991.

[3] B. R. Badrinath, A. Acharya, and T. Imielinski. Structuring Distributed Algorithms for Mobile Hosts. In *Proceedings of the 14^{th} International Conference on Distributed Computing Systems (to appear)*, June 1994.

[4] P. Bhagwat and C. E. Perkins. A Mobile Networking System Based on Internet Protocol(IP). In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 69–82, August 1993.

[5] S. Chandrasekaran and S. Venkatesan. A Message-Optimal Algorithm for Distributed Termination Detection. *Journal of Parallel and Distributed Computing*, pages 245–252, 1990.

[6] K. M. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[7] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[8] J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11^{th} Australian Computer Science Conference*, pages 56–66, February 1988.

[9] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, April 1994.

[10] S. T. Huang. Detecting Termination of Distributed Computations by External Agents. In *Proceedings of the 9^{th} International Conference on Distributed Computing Systems*, pages 79–84, 1989.

[11] J. Ioannidis, D. Duchamp, and G. Q. Maguire. IP-based Protocols for Mobile Internetworking. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, pages 235–245, 1991.

[12] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[13] T.-H. Lai and T.-H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25:153–158, 1987.

[14] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] P.-J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *Proceedings of the* 4^{th} *International Conference on Data Engineering*, pages 154–163, February 1988.

[16] F. Mattern. Virtual Time and Global States of Distributed Systems. In M.Cosnard et. al., editor, *Proceedings of the Workshop on Parallel and Distributed Algorithm*, pages 215–226. Elsevier Science Publishers B.V.(North-Holland), 1989.

[17] F. Mattern. Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non-FIFO Systems. Technical Report SFB124–24/90, University of Kaiserslautern, 1990.

[18] R. Prakash and M. Singhal. Maximal Global Snapshot with Concurrent Initiators. Technical Report OSU–CISRC–3/94–TR12, The Ohio State University, 1994.

[19] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 223–238, 1989.

[20] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[21] F. Teraoka, Y. Yokote, and M. Tokoro. A Network Architecture Providing Host Migration Transparency. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, 1991.

[22] S. Venkatesan. Message-Optimal Incremental Snapshots. *Journal of Computer and Software Engineering*, 1(3):211–231.

[23] S. Venkatesan. Optimistic Crash Recovery Without Rolling Back Non-Faulty Processors. accepted for Information Sciences — An International Journal, 1993.

[24] S. Venkatesan and Tony T.-Y. Juang. Low Overhead Optimistic Crash Recovery. Preliminary version appears in Proceedings of 11^{th} International Conference on Distributed Computing Systems as *Crash Recovery with Little Overhead*, pp 454–461, 1991.

[25] H. Wada, T. Yozawa, T. Ohnishi, and Y. Tanaka. Mobile Computing Environment Based on Internet Packet Forwarding. In *1991 Winter USENIX*, 1993.