Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems *

Guohong Cao and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
E-mail: {gcao, singhal}@cis.ohio-state.edu

Abstract

Mobile computing raises many new issues, such as lack of stable storage, low bandwidth of wireless channel, high mobility, and limited battery life. These new issues make traditional checkpointing algorithms unsuitable. Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications, since it avoids domino effects and minimizes the stable storage requirement. However, it suffers from high overhead associated with the checkpointing process in mobile computing systems. Two approaches have been used to reduce the overhead: first is to minimize the number of synchronization messages and the number of checkpoints; the other is to make the checkpointing process non-blocking. These two approaches were orthogonal previously until the Prakash-Singhal algorithm [25] combined them. However, we [5] found some problems in their algorithm and proved that there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints. In this paper, we introduce the concept of "mutable checkpoint", which is neither a tentative checkpoint nor a permanent checkpoint, to design an efficient checkpointing algorithm for mobile computing systems. Mutable checkpoints can be saved anywhere; e.g., the main memory or local disk of MHs. In this way, taking a mutable checkpoint avoids the overhead of transferring large amount of data to the stable storage at MSSs over the wireless network. We present techniques to minimize the number of mutable checkpoints. Simulation results show that the overhead of taking mutable checkpoints is negligible. Based on mutable checkpoints, our non-blocking algorithm avoids the avalanche effect and forces only a minimum number of processes to take their checkpoints on stable storage.

Key words: Mobile computing, coordinated checkpointing, causal dependency, non-blocking.

*This is a substantially enhanced version of a paper presented at ICDCS'98.

1 Introduction

A distributed system is a collection of processes that communicate with each other by exchanging messages. A mobile computing system is a distributed system where some of processes are running on *mobile hosts* (MHs), whose location in the network changes with time. To communicate with mobile computers, a conventional distributed system is augmented with *mobile support stations* (MSSs) that act as access points for the mobile computers by wireless networks.

The mobility of MHs raises some new issues [1, 18] that complicate the design of checkpointing algorithms:

- Changes in the location of an MH complicate the routing of messages. Messages sent by an MH to another MH may have to be rerouted because the destination MH has moved. Locating an MH generally increases the delay and message complexity.
- Due to the vulnerability of mobile computers to catastrophic failures, e.g., loss, theft, or physical damage, the disk storage on an MH cannot be considered as the stable storage. A reasonable solution [1] is to utilize the stable storage at MSSs to store checkpoints of MHs. Thus, to take a checkpoint, an MH has to transfer large amount of data to its local MSS over the wireless network. Since the wireless network has low bandwidth and MHs have relatively low computation power, a checkpointing algorithm should only force minimum number of processes to take checkpoints.
- The battery at an MH has limited life. To save energy, the MH can power down individual components during periods of low activity [12]. This strategy is referred to as the *doze mode* operation. An MH in the doze mode is woken up on receiving a message. Therefore, energy conservation and low bandwidth constraints require a checkpointing algorithm to minimize the number of synchronization messages.
- MHs may disconnect from the network temporarily or permanently. The disconnection of MHs should not prevent the checkpointing process.

These constraints make traditional checkpointing algorithms for distributed systems unsuitable for mobile computing systems.

Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications since it avoids the domino effect [17] and minimizes the stable storage requirement. In this approach, the state of each process in the system is periodically saved on stable storage, which is called a checkpoint of the process. To recover from a failure, the system restarts its execution from a previous consistent global checkpoint saved on stable storage. A system state is said to be consistent if it contains no *orphan message*; i.e., a message whose receive event is recorded in the state of the destination process, but its send event is lost [17, 29]. In order to record a consistent global checkpoint, processes must synchronize their checkpointing activities. In other words, when a process takes a checkpoint, it asks (by sending checkpoint requests to) all relevant processes to take checkpoints. Therefore, coordinated checkpointing suffers from high overhead associated with the checkpointing process.

Much of the previous work [10, 16, 17, 21] in coordinated checkpointing has focused on minimizing the number of synchronization messages and the number of checkpoints during checkpointing. However, these algorithms (called *blocking algorithms*) force all relevant processes in the system to block their computations during the checkpointing process. Checkpointing includes the time to trace the dependency tree and to save the states of processes on stable storage, which may be long. Moreover, in mobile computing systems, due to the mobility of MHs, a message may be routed several times before reaching its destination. Therefore, blocking algorithms may further degrade the performance of mobile computing systems [4, 11].

Recently, nonblocking algorithms [11, 27] have received considerable attention. In these algorithms, processes need not block during checkpointing by using a checkpointing sequence number to identify orphan messages. However, these algorithms [11, 27] require all processes in the computation to take checkpoints during the checkpointing, even though many of them may not be necessary.

Prakash-Singhal algorithm [25] was the first algorithm to combine these two approaches. More specifically, it only forces minimum number of processes to take checkpoints and does not block the underlying computation during the checkpointing. However, we found that this algorithm may result in an inconsistency [5] in some situations and we proved that there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints.

In this paper, we introduce the concept of "mutable checkpoint", which is neither a tentative checkpoint nor a permanent checkpoint, to design an efficient checkpointing algorithm for mobile computing systems. Mutable checkpoints need not be saved on stable storage and it can be saved anywhere; e.g., the main memory or local disk of MHs. Thus, taking a mutable checkpoint avoids the overhead of transferring large amount of data to the stable storage in MSS over the wireless network. We present techniques to minimize the number of mutable checkpoints. Simulation results show that the overhead of taking mutable checkpoints is negligible. Based on mutable checkpoints, our non-blocking algorithm avoids the avalanche effect and forces only a minimum number of processes to take their checkpoints on stable storage.

The rest of the paper is organized as follows. Section 2 develops the necessary background. In Section 3, we present a low-cost checkpointing algorithm for mobile computing systems. The correctness proof is provided in Section 4. In Section 5, we evaluate the performance of our algorithm. Related work is provided in Section 6. Section 7 concludes the paper.

2 Preliminaries

2.1 Computation Model

A mobile computing system consists of a large number of *mobile hosts* (MHs) [1] and relatively fewer static hosts called *mobile support stations* (MSSs). The MSSs are connected by a static wired network, which provides reliable $FIFO$ delivery of messages. A *cell* is a logical or geographical area covered by an MSS. An MH can directly communicate with an MSS by a reliable $FIFO$ wireless channel only if it is present in the cell supported by the MSS.

The distributed computation we consider consists of N sequential processes denoted by $P_0, P_1, P_2, \cdots, P_N$ running concurrently on fail-stop MHs or MSSs in the network. The processes do not share a common memory or a common clock. Message passing is the only way for processes to communicate with each other. The computation is asynchronous: each process progresses at its own speed and messages are exchanged through reliable communication channels, whose transmission delays are finite but arbitrary. The messages generated by the underlying distributed application will be referred to as the *computation messages*. Messages generated by the processes to advance checkpoints will be referred to as the *system messages*.

Each checkpoint taken by a process is assigned a unique sequence number. The $i^{th}(i \geq 0)$ checkpoint of process P_p is assigned a sequence number i and is denoted by $C_{p,i}$. The i^{th} *checkpoint interval* [23] of process P_p denotes all the computation performed between its i^{th} and $(i+1)^{th}$ checkpoint, including the i^{th} checkpoint but not the $(i+1)^{th}$ checkpoint.

2.2 Handling Mobility and Disconnections

Changes in the location of an MH complicate the routing of messages. Messages sent by an MH to another MH may have to be rerouted because the destination MH disconnected from the old MSS and is now connected to a new MSS. Many routing protocols for the network layer have been proposed [14, 30] to handle MH mobility.

It should be noted that disconnection of a MH is a voluntary operation [1], and frequent disconnections of MHs is an expected feature of the mobile computing environment. Unexpected disconnections due to battery failure, processor failure, or network failure are different from voluntary disconnection and are discussed in Section 3.6.

An MH may get disconnected from the network for an arbitrary period of time. At the application level, the checkpointing algorithm may generate a request for the disconnected MH to take a checkpoint. Delaying a response to such a request until the MH reconnects at some MSS, may significantly increase the completion time of the checkpointing algorithm. So, we propose the following solution to deal with disconnections.

Note that only local events can take place at an MH during the disconnect interval. No message send or receive event occurs during this interval. Hence, no new dependencies with respect to other processes are created during this interval. The dependency relation of an MH with the rest of the system, as reflected by its local checkpoint, is the same no matter when the local checkpoint is taken during the disconnect interval.

Suppose a mobile host MH_i wants to disconnect from its local MSS_p. MH_i takes a local checkpoint and transfers its local checkpoint to MSS_p as $disconnect\_checkpoint_i$. If MH_i is asked to take a checkpoint during the disconnect interval, MSS_p converts $disconnect\_checkpoint_i$ into MH_i's new checkpoint, and uses the message dependency information of MH_i to propagate the checkpoint request. MH_i also sends $disconnect(sn)$ message to MSS_p on the MH-to-MSS channel supplying the sequence number sn of the last message received on the MSS-to-MH channel. On the receipt of MH_i's $disconnect(sn)$, MSS_p knows the last message that MH_i received from it and buffers all computation messages received until the end of the disconnect interval.

Later, suppose MH_i reconnects at an MSS, say MSS_q. If MH_i knows the identity of its last MSS, say MSS_p, it sends a $reconnect(MH_i, MSS_q)$ message to MSS_p through MSS_q. If MH_i lost the identity of its last MSS for some reason, then MH_i's *reconnect* request is broadcasted over the network. On receiving the *reconnect* request, MSS_p transfers all the support information (the checkpoint, dependency vector, buffered messages, etc.) of MH_i to MSS_q, and deletes all information related to the disconnection. Then, MSS_q forwards all the support information to MH_i. When the data sent by MSS_p arrives at MH_i, MH_i processes the buffered messages. If MSS_p has taken a checkpoint for MH_i, MH_i clears its message dependency information before processing the buffered messages. With this, the reconnect routine terminates and the relocated mobile host MH_i resumes normal communication with other MHs (or MSSs) in the system.

2.3 Basic Idea Behind Non-blocking Algorithms

Most existing coordinated checkpointing algorithms [10, 17, 21] rely on the two-phase protocol and save two kinds of checkpoints on stable storage: *tentative* and *permanent*. In the first phase, the initiator takes a tentative checkpoint and forces all relevant processes to take tentative checkpoints. Each process informs the initiator whether it succeeded in taking a tentative checkpoint. A process may refuse to take a checkpoint depending on its underlying computation. After the initiator has received positive reply from all relevant processes, the algorithm enters the second phase. If the initiator learns that all processes have successfully taken tentative checkpoints, it asks them to make their tentative checkpoints permanent; otherwise, it asks them to discard their tentative checkpoints. A process, on receiving the message from the initiator acts accordingly. Note that after a process takes a tentative checkpoint in the first phase, it remains blocked until it receives the decision from the initiator in the second phase.

A non-blocking checkpointing algorithm does not require any process to suspend its underlying computation. When processes do not suspend their computations, it is possible for a process to receive a computation message from another process, which is already running in a new checkpoint interval. If this situation is not properly dealt with, it may result in an inconsistency. For example, in Figure 1, P_2 initiates a checkpointing process. After sending checkpoint requests to P_1 and P_3, P_2 continues its computation. P_1 receives the checkpoint request and takes a new checkpoint, then it sends $m1$ to P_3. Suppose P_3 receives the checkpoint request from P_2 after receiving $m1$, $m1$ becomes an orphan (see Figure 1).

Most non-blocking algorithms [11, 22, 27] use a Checkpoint Sequence Number (csn) to avoid inconsistencies. More specifically, a process is forced to take a checkpoint if it receives a computation message whose csn is greater than its local csn. In Figure 1, P_1 increases its

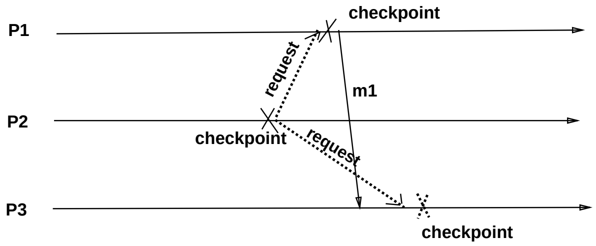

Figure 1: Inconsistent checkpoints

csn after it takes a checkpoint and appends the new csn to $m1$. When P_3 receives $m1$, it takes a checkpoint before processing $m1$ because the csn appended to $m1$ is larger than its local csn.

This scheme works only when every process in the computation can receive each checkpoint request and then increases its own csn. Since Prakash-Singhal algorithm [25] only forces a part of processes to take checkpoints, the csn of some processes may be out-of-date, and may not be able to avoid inconsistencies. Prakash-Singhal algorithm attempts to solve this problem by having each process maintain an array to save the csn, where $csn[i]$ is the expected csn of P_i. Note that P_i's $csn_i[i]$ may be different from P_j's $csn_j[i]$ if there has been no communication between P_i and P_j for several checkpoint intervals. By using csn and the initiator identification number, they claim that their non-blocking algorithm can avoid inconsistencies and minimize the number of checkpoints during checkpointing. However, we showed that this algorithm may result in an inconsistency [5] and we have proved that there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints [5]. Since the proof is not the major concern of this paper, we only briefly mention the basic idea using an example.

2.4 Impossibility of Checkpointing

In Figure 2, assume messages $m6$ and $m7$ do not exist. To initiate a checkpointing process, P_1 takes checkpoint $C_{1,1}$ and sends checkpoint requests to P_3 and P_4 (not illustrated in the figure) since it depends on them. When P_4 receives the checkpoint request, it takes a checkpoint and sends a checkpoint request to P_5. For the same reason, P_5 takes a checkpoint and sends a checkpoint request to P_2. P_2 must take this checkpoint before processing $m5$; otherwise, $m5$ will become an orphan. If $m4$ does not exist, P_2 will not receive a checkpoint request associated with checkpoint $C_{1,1}$. In this case, P_2 should not take a checkpoint before processing $m5$ in order to minimize the checkpoints. Therefore, when P_2 receives $m5$, it has to decide whether to take a checkpoint before processing $m5$. In other words, P_2 has to know if it will receive a checkpoint request associated with $C_{1,1}$ in the future when it receives $m5$. However, if the checkpointing process is non-blocking, there is not enough information for P_2 to look into the future.

The problem arises due to the dependency created by messages, e.g., $m4$. Because of $m4$, there is a new dependency between P_1 and P_2 such that P_2 will receive a checkpoint request associated with $C_{1,1}$. There are two possible approaches for P_2 to get the information about this new dependency (called the *z-dependency* in [5]).

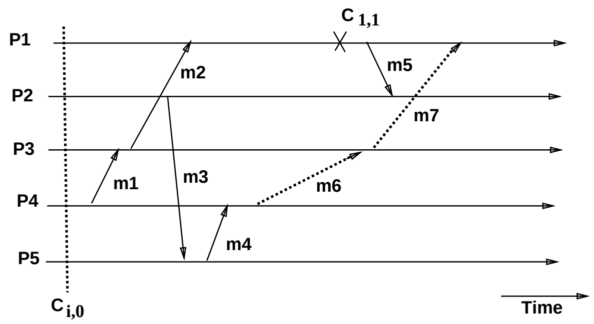

Figure 2: Tracing the dependency

Approach 1 (Tracing the in-coming messages): In this approach, P_2 gets the new z-dependency information from P_1. Then, P_1 has to know the z-dependency information before it sends $m5$ and appends the z-dependency information to $m5$. In Figure 2, P_1 cannot get the new z-dependency information unless P_4 notifies P_1 of the new z-dependency information when P_4 receives $m4$. There are two ways for P_4 to notify P_1 of the new z-dependency information: first is to broadcast the z-dependency information (not illustrated in the figure); the other is to send the z-dependency information on an extra message $m6$ to P_3, which in turn sends it to P_1 on $m7$. Both of them dramatically increase message overhead. Since the algorithm does not block the underlying computation, it is possible that P_1 receives $m7$ after it sends out $m5$ (as shown in the figure). Then, P_2 is still not guaranteed to get the z-dependency information when it receives $m5$.

Approach 2 (Tracing the out-going messages): In this approach, since P_2 sends message $m3$ to P_5, P_2 hopes to get the new z-dependency information from P_5. Then, P_5 has to know the new z-dependency information and it must send an extra message (not shown in the figure) to notify P_2. Similarly, P_5 needs to get the new z-dependency information from P_4, which comes from P_3, and finally from P_1. Certainly, this requires much more extra messages than Approach 1. Similar to Approach 1, P_2 is still not guaranteed to get the z-dependency information in time since the computation is in progress.

In conclusion, there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints.

3 A Checkpointing Algorithm

In this section, we present our checkpointing algorithm, which neither blocks the underlying computation nor forces all processes to take checkpoints.

3.1 Basic Idea

Basic Scheme

A simple non-blocking scheme for checkpointing is as follows: when a process P_i sends a message, it piggybacks the current value of $csn_i[i]$ (csn is explained in Section 2.3). When a process P_j receives a message m from P_i, P_j processes the message if $m.csn \leq csn_j[i]$; otherwise, P_j takes a checkpoint, updates its csn ($csn_j[i] = m.csn$), and then processes the message. This method may result in a large number of checkpoints. Moreover, it may lead to an *avalanche effect*, in which processes in the system recursively ask others to take checkpoints.

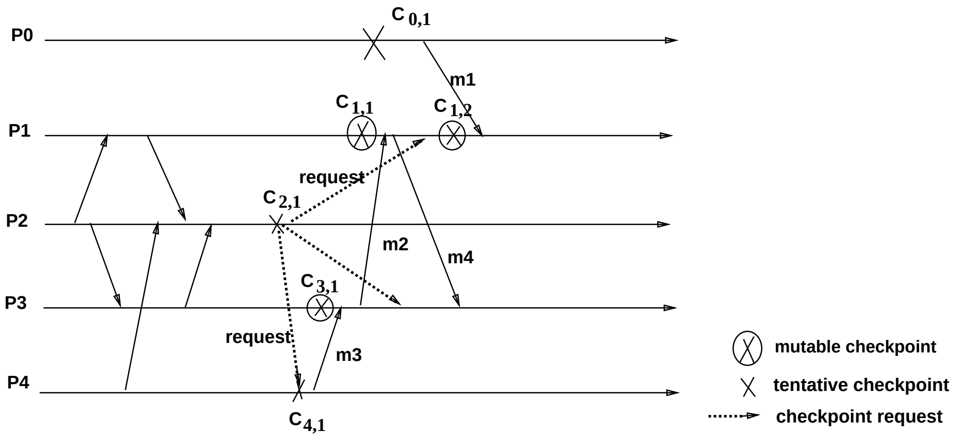

Figure 3: An example of checkpointing

For example, in Figure 3, to initiate a checkpointing process, P_2 takes its own checkpoint and sends checkpoint requests to P_1, P_3 and P_4. When P_2's request reaches P_4, P_4 takes a checkpoint. Then, P_4 sends message $m3$ to P_3. When $m3$ arrives at P_3, P_3 takes a checkpoint before processing it because $m3.csn > csn_3[4]$. For the same reason, P_1 takes a checkpoint before processing $m2$.

P_0 has not communicated with other processes before it takes a local checkpoint. Later, it sends a message $m1$ to P_1. P_1 takes checkpoint $C_{1,2}$ before processing $m1$, because P_0 has taken a checkpoint with checkpoint sequence number larger than P_1 expected. Then, P_1 requires P_3 to take another checkpoint (not shown in the figure) due to $m2$ and P_3 in turn asks P_4 to take another checkpoint (not shown in the figure) due to $m3$. If P_4 had received messages from other processes after it sent $m3$, then those processes would have been forced to take checkpoints. This chain may never end.

We reduce the number of checkpoints based on the following observation. In Figure 3, if $m4$ does not exist, it is not necessary for P_1 to take $C_{1,2}$ since checkpoint $C_{1,1}$ is consistent with the rest of checkpoints. Based on this observation, we get the following revised scheme.

When a process P_j receives a message m from P_i, P_j only takes a checkpoint when $m.csn > csn_j[i]$ and P_j has sent at least one message in the current checkpoint interval.

In Figure 3, if $m4$ does not exist, $C_{1,2}$ is not necessary according to the revised scheme. However, if $m4$ exists, the revised scheme still results in a large number of checkpoints and may result in an avalanche effect.

Enhanced Scheme

We now present the basic idea of our scheme that eliminates avalanche effects during checkpointing. From Figure 3, we make two observations:

Observation 1: It is not necessary to take checkpoint $C_{1,2}$ even though $m4$ exists, since P_1 will not receive a checkpoint request associated with $C_{0,1}$. Note that $m4$ will not become an orphan even though it does not take checkpoint $C_{1,2}$.

Observation 2: From Section 2.4, P_1 does not have enough information to know if it will receive a checkpoint request associated with $C_{0,1}$ when P_1 receives $m1$.

These observations imply that $C_{1,2}$ is unnecessary but still unavoidable. Thus, there are two kinds of checkpoints in response to computation messages. In Figure 3, $C_{1,1}$ is different from $C_{1,2}$. $C_{1,1}$ is a checkpoint associated with initiator P_2, and P_1 will receive a checkpoint request for this checkpointing initiated by P_2. $C_{1,2}$ is a checkpoint associated with initiator P_0, but P_1 will not receive a checkpoint request for this checkpointing initiated by P_0 in the future. To avoid inconsistency, P_1 should keep $C_{1,1}$ when it receives P_2's request. However, P_1

can discard $C_{1,2}$ after the checkpointing initiated by P_0 terminates ($C_{0,1}$ becomes permanent checkpoint) since at that time, P_1 is sure that it will not receive any checkpoint request associated with P_0's initiation. Moreover, if P_0 has finished its checkpointing process before it sends $m1$, P_1 does not need to take checkpoint $C_{1,2}$.

We introduce a new concept, called *mutable checkpoint*, to reflect the essence of checkpoints (like $C_{1,1}, C_{1,2}$) triggered by computation messages. A mutable checkpoint is neither a tentative checkpoint nor a permanent checkpoint, but it can be turned into a tentative checkpoint. When a process takes a mutable checkpoint, it does not send checkpoint requests to other processes and it does not need to save the checkpoint on stable storage. It can save the mutable checkpoint anywhere; e.g., in the main memory or the local disk of MHs. Suppose a process P_i has taken a mutable checkpoint. When P_i receives a checkpoint request, it transfers the mutable checkpoint to the stable storage and forces all dependent processes to take tentative checkpoints. In this way, P_i turns its mutable checkpoint into a tentative checkpoint. If P_i does not receive a checkpoint request after the checkpointing activity terminates (implementation details will be discussed in the next section), it discards the mutable checkpoint.

In Figure 3, when $m2$ arrives at P_1, P_1 takes a mutable checkpoint $C_{1,1}$ before processing it because $m2.csn > csn_1[3]$. $C_{1,1}$ is turned into a tentative checkpoint when P_1 receives the checkpoint request sent by P_2. If P_0 has finished its checkpointing activity before it sends $m1$, P_1 does not need to take a mutable checkpoint $C_{1,2}$. Otherwise, P_1 takes a mutable checkpoint $C_{1,2}$, which will be discarded when P_0's checkpointing terminates. Since $C_{1,2}$ is a mutable checkpoint, it does not force P_3 to take a new checkpoint. Thus, our scheme avoids the avalanche effect and significantly reduces the checkpointing overhead. If there is no ambiguity, we simply refer to a tentative or permanent checkpoint as a checkpoint.

Further reduction in the number of checkpoints

In the above scheme, a process may receive unnecessary checkpoint requests and may take unnecessary checkpoints. As shown in Figure 4, P_2 initiates a checkpointing process by taking a checkpoint $C_{2,1}$ and forces P_1 to take a checkpoint $C_{1,1}$ (due to $m2$). Later, to initiate a checkpointing process, P_3 takes a checkpoint $C_{3,1}$ and sends a request to P_2 due to $m1$. When P_2 receives the request, it takes a checkpoint $C_{2,2}$ and forces P_1 to take a checkpoint $C_{1,2}$. However, $C_{2,2}$ and $C_{1,2}$ are not necessary since $m1$ is not an orphan even though $C_{1,2}$ and $C_{2,2}$ do not exist.

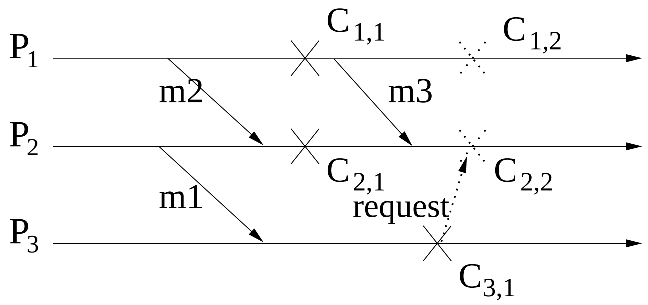

Figure 4: Further reduce the number of checkpoints

These unnecessary checkpoints can be avoided by the following method. When a process P_i sends a checkpoint request to P_j, it attaches $csn_i[j]$ to the request. On receiving the request, P_j compares the attached $csn_i[j]$ (req_csn) with its own $csn_j[j]$. If $csn_j[j] > req\_csn$ (i.e., P_j has recorded the sending of the message which creates the dependency between P_i and P_j), P_j does not need to take a checkpoint; otherwise, it takes a checkpoint. In Figure 4, when P_3 sends a request to P_2, it attaches $csn_3[2] = 0$ to the request. When P_2 receives the request, $csn_2[2]$ has been increased to 1 due to $C_{2,1}$. Thus, P_2 ignores this request and does not take checkpoint $C_{2,2}$, and subsequently P_2 does not force P_1 to take checkpoint $C_{1,2}$.

3.2 Data Structures

The following data structures are used in our algorithm:

R_i: an array of n bits at process P_i. $R_i[j] = 1$ represents that P_i receives a computation message from P_j in the current checkpoint interval.

csn_i: an array of n checkpoint sequence numbers (csn) at each process P_i. $csn_i[j]$ represents the checkpoint sequence number of P_j that P_i knows. In other words, P_i expects to receive a message from P_j with checkpoint sequence number $csn_i[j]$.

$weight$: a non-negative variable of type real with maximum value of 1. It is used to detect the termination of the checkpointing as in [13].

$trigger_i$: a tuple (pid, $inum$) maintained by each process P_i. pid indicates the checkpoint initiator that triggered the latest checkpointing process. $inum$ indicates the csn at process pid when it took its own local checkpoint on initiating the checkpointing.

$sent_i$: a boolean, which is set to 1 if P_i has sent a message in the current checkpoint interval.

cp_state_i: a boolean, which is set to 1 if P_i is during the checkpointing process.

old_csn: a variable used to save the csn of the current tentative (permanent) checkpoint.

CP_i: a record maintained by each process P_i. Each record has the following fields:

mutable: the mutable checkpoint of P_i.
R: P_i's own boolean vector before it takes the current mutable checkpoint.
trigger: the *trigger* which is associated with the current mutable checkpoint.
sent: P_i's own *sent* before it takes the current mutable checkpoint.

csn is initialized to an array of 0's at all processes. The trigger tuple at process P_i is initialized to $(i, 0)$. The *weight* and *cp_state* at a process is initialized to 0. When a process P_i sends a computation message, it appends its $csn_i[i]$ to the message. Also, P_i checks if cp_state_i is equal to 1. If so, it appends its trigger to the computation message.

When a process P_j receives a checkpoint request from P_i, we say "P_j *inherits* a request from P_i" if only if $old\_csn_j \leq req\_csn$ (req_csn is appended with the request) and P_j takes a tentative checkpoint. In this definition, we use old_csn_j instead of $csn_j[j]$ used in Section 3.1, since $csn_j[j]$ is also increased when taking a mutable checkpoint, but we need to compare req_csn with the csn of the current tentative (permanent) checkpoint.

3.3 The Checkpointing Algorithm

In this section, we present our non-blocking checkpointing algorithm. To clearly present the algorithm, we assume that at any time, at most one checkpointing is in progress. In Section 3.5, we extend the algorithm for concurrent invocations.

Checkpointing initiation: Any process can initiate a checkpointing process. When a process P_i initiates a checkpointing process, it takes a local checkpoint, increments its $csn_i[i]$, sets $weight_i$ to 1, sets cp_state_i to 1, and stores its own identifier and the new $csn_i[i]$ in its trigger. Then, it sends a checkpoint request to each process P_j such that $R_i[j]$=1 and resumes its computation. Each request carries the trigger of the initiator, R_i, and a portion of the weight of the initiator, whose weight is decreased by an equal amount.

Reception of a checkpoint request: When a process P_i receives a request from P_j, it first compares req_csn with its old_csn to see if it needs to inherit the request. If P_i does not need to inherit the request, it sends the appended *weight* to the initiator and then exits. Otherwise, it updates its csn and *cp_state*, and compares $P_j.trigger$ ($msg\_trigger$) with $P_i.trigger$ ($own\_trigger$). If $msg\_trigger = own\_trigger$ (implying that P_i has already taken a checkpoint for this checkpointing), P_i checks if there is a mutable checkpoint which has a trigger identical to $msg\_trigger$. If not, P_i sends the appended *weight* to the initiator; otherwise, P_i saves the mutable checkpoint on stable storage (the mutable checkpoint is turned into a tentative checkpoint), and then propagates the request. If P_i propagates the request to all processes on which it depends, it may result in a large number of redundant system messages, since some processes on which P_i depends may have received the request from other processes. The Koo-Toueg algorithm [17] uses this approach, and its system message overhead can be as large as $O(N^2)$, where N is the number of processes in the system. On the other hand, only propagating the request to processes on which P_i depends, but P_j (the sender) does not, may not work, since a process receives a request does not necessarily mean that it inherits the request. We solve this problem by attaching some information (csn and R which are saved in MR) to the request. Then, P_i only propagates the request to each process P_k on which P_i depends, but P_k may have not inherited the request; that is, if P_i knows (by MR) some other process has sent the request to P_k with $req\_csn \geq csn_i[k]$ (req_csn is appended with the request and saved in $MR[k].csn$), it does not need to send the request to P_k; otherwise, it has to send the request since P_k may inherit the request from P_i. Also, P_i appends the initiator's trigger and a portion of the received weight to all those requests. At last, P_i sends a *reply* to the initiator with the weight equal to the remaining weight and resumes its underlying computation. If $msg\_trigger \neq own\_trigger$, P_i takes a tentative checkpoint, increases its $csn_i[i]$, and propagates the request as above. Then, P_i clears R_i and $sent_i$, sends a *reply* to the initiator with the remaining weight, and then resumes its underlying computation.

Computation messages received during checkpointing: When P_i receives a computation message from P_j, P_i compares $m.csn$ with its local $csn_i[j]$. If $m.csn \leq csn_i[j]$, the message is processed and no checkpoint is taken. Otherwise, it implies that P_j has taken a checkpoint before sending m. P_i updates its $csn_i[j]$ to $m.csn$ and checks if the following conditions are satisfied:

Condition 1: P_j is in checkpointing process before sending m.

Condition 2: P_i has sent a message since last checkpoint.

Condition 3: P_i has not taken a checkpoint associated with the initiator (in the msg_trigger).

If all of them are satisfied, P_i takes a mutable checkpoint and updates its data structures, such as csn, CP, R, cp_state, and $sent$. If only Condition 1 is satisfied, P_i only increases $csn_i[i]$ and sets cp_state_i to 1.

Termination and garbage collection: The initiator adds weights received in all *reply* messages to its own *weight*. When its weight becomes equal to 1, it concludes that all processes involved in the checkpointing have taken their tentative checkpoints. Then, it broadcasts *commit* messages to all processes in the system. If a process has taken a tentative checkpoint, on receiving the *commit* message, it makes its tentative checkpoint permanent and clears *cp_state*. Other processes also clear their *cp_state* and discard mutable checkpoints if there is any. Note that when a process discards its mutable checkpoints, it updates its R and *sent*.

Instead of broadcasting *commit* messages to all processes. In [6], the initiator only sends *commit* messages to those processes from which it has received *reply* messages. However, to clear *cp_state*, each process needs to maintain a history of the processes to which it has sent messages when its *cp_state* is equal to 1. Also, it notifies them to clear their *cp_state*. There is a tradeoff between these two approaches. If there are many communications among processes during last checkpoint interval, the broadcast approach is better. On the other hand, if there are only a limited number of message exchanges during last checkpoint interval, the update approach [6] is better. To obtain the advantages of both approaches, the initiator can use a counter to save the number of processes that have taken checkpoints. If the counter is larger than a value (a system tuning parameter), the broadcast approach is used; otherwise, the update approach is used. Since this paper concentrates on reducing the overhead of saving checkpoints, we simply use the broadcast approach.

A formal description of the checkpointing algorithm is given below:

Actions taken when P_i sends a computation message to P_j:
if $cp\_state_i = 1$
then send(P_i, message, $csn_i[i], own\_trigger$); $sent_i := 1$;
else send(P_i, message, $csn_i[i], NULL$); $sent_i := 1$;

Actions for the initiator P_j:
increment($csn_j[j]$); $own\_trigger := (P_j, csn_j[j])$; $cp\_state_j := 1$;
for k := 0 to N **do** $MR[k].csn := 0$; $MR[k].R := 0$;
$MR[j].csn := csn_j[j]$; $MR[j].R := 1$;
prop_cp($R_j, MR, P_j, own\_trigger$, 1.0);
take a local checkpoint (on stable storage); $old\_csn_j := csn_j[j]$; $sent_j := 0$; reset R_j;

Actions at process, P_i, on receiving a checkpoint request from P_j:
receive($P_j, request, MR, recv\_csn, msg\_trigger, req\_csn, recv\_weight$);
$csn_i[j] := recv\_csn$;
if $old\_csn_i > req\_csn$
then send($P_i, reply, recv\_weight$) to the initiator; return;
$cp\_state_i := 1$;
if $msg\_trigger = own\_trigger$
then if $CP_i.trigger = msg\_trigger$

then prop_cp($CP_i.R, MR, P_i, msg\_trigger, recv\_weight$);
save $CP_i.mutable$ on stable storage; $old\_csn_i := csn_i[i]$; $CP_i := NULL$;
send($P_i, reply, weight_i$) to the initiator;
else send($P_i, reply, recv\_weight$) to the initiator;
else increment($csn_i[i]$); $own\_trigger := msg\_trigger$;
prop_cp($R_i, MR, P_i, msg\_trigger, recv\_weight$);
take a local checkpoint (on stable storage); $old\_csn_i := csn_i[i]$;
send($P_i, reply, weight_i$) to the initiator;
$sent_i := 0$; reset R_i;

Actions at process P_i, on receiving a computation message from P_j:
receive($P_j, m, recv\_csn, msg\_trigger$);
if recv_csn $\leq csn_i[j]$
then $R_i[j] := 1$; process the message;
else if $csn_i[msg\_trigger.pid] = msg\_trigger.inum$
then $csn_i[j] := recv\_csn$; $R_i[j] := 1$; process the message;
else $csn_i[j] := recv\_csn$;
if msg_trigger $\neq NULL \wedge sent_i = 1 \wedge$ $msg\_trigger \neq own\_trigger$
then take a local checkpoint, save it in $CP_i.mutable$;
$CP_i.trigger := msg\_trigger$; $CP_i.R := R_i$; $CP_i.sent := sent_i$;
$sent_i := 0$; reset R_i;
if $msg\_trigger \neq NULL \wedge cp\_state_i = 0$
then $cp\_state_i := 1$; increment($csn_i[i]$); $own\_trigger := msg\_trigger$;
$R_i[j] := 1$; process the message;

prop_cp($R_i, MR, P_i, msg\_trigger, recv\_weight$)
$weight_i := recv\_weight_i$;
for k := 0 to N **do** temp[k].csn := $max(MR[k].csn, csn_i[k])$; temp[k].R := $max(MR[k].R, R_i[k])$;
for any P_k, such that $(R_i[k] = 1) \wedge (max(MR[k].csn, csn_i[k]) \neq MR[k].csn)$
$weight_i := weight_i/2$; send($P_i, request$, temp, $csn_i[i], msg\_trigger, csn_i[k], weight_i$);

Actions in the second phase for the initiator P_i:
receive($P_j, reply, recv\_weight$); $weight_i := weight_i + recv\_weight$;
if $weight_i = 1$
then $cp\_state_i := 0$; broadcast($commit, msg\_trigger$);

Actions at other process P_j on receiving a broadcast message:
receive($commit, msg\_trigger$); $csn_j[msg\_trigger.pid] = msg\_trigger.inum$;
$cp\_state_j := 0$;
if $CP_j.trigger = msg\_trigger \wedge CP_j \neq NULL$
then $sent_j := sent_j \cup CP_j.sent$; $R_j := R_j \cup CP_j.R$; $CP_j := NULL$;
if there is a tentative checkpoint associated with $msg\_trigger$, make it permanent;

3.4 An Example

The basic idea of the algorithm can be better understood by the example in Figure 3. To initiate a checkpointing process, P_2 takes its own checkpoint and sends checkpoint requests to P_1, P_3 and P_4, because $R_2[1] = 1, R_2[3] = 1$, and $R_2[4] = 1$. When P_2's request reaches P_4, P_4 takes a checkpoint. Then, P_4 sends message $m3$ to P_3. When $m3$ arrives at P_3, P_3 takes a mutable checkpoint before processing it because $m3.csn > csn_3[4]$ and P_3 has sent a message during the current checkpoint interval. For the same reason, P_1 takes a mutable checkpoint before processing $m2$.

P_0 has not communicated with other processes before it takes a local checkpoint. Later, it sends a message $m1$ to P_1. If P_0 has finished its checkpointing process before it sends $m1$, P_1 does not need to take the checkpoint $C_{1,2}$. Otherwise, P_1 takes a mutable checkpoint $C_{1,2}$ before processing $m1$.

When P_1 receives the checkpoint request from P_2, since $C_{1,1}$ is a mutable checkpoint associated with P_2, P_1 turns $C_{1,1}$ into a tentative checkpoint by saving it on stable storage. Similarly, P_3 converts $C_{3,1}$ to a tentative checkpoint when it receives the checkpoint request from P_2. Finally, the checkpointing initiated by P_2 terminates when checkpoints $C_{1,1}, C_{2,1}, C_{3,1}$, and $C_{4,1}$ are made permanent. P_1 discards $C_{1,2}$ when it makes checkpoint $C_{1,1}$ permanent or receives P_0's *commit*, whichever is earlier.

3.5 Multiple concurrent initiations

The simplest way to handle concurrent checkpoint initiations is to use the techniques in [17]. When a process P_i receives a checkpoint request from P_j while executing the checkpoint algorithm, P_i ignores P_j's checkpoint request or defers the request until it finishes its current checkpointing. If P_i's checkpoint request is ignored by a process, P_i has to abort its checkpointing efforts, which results in poor performance. A more efficient technique to handle concurrent checkpoint initiations can be found in [24]. As multiple concurrent checkpoint initiation is orthogonal to our discussion, we only briefly mention the main features of [24]. When a process receives its first request for checkpointing initiated by another process, it takes a local checkpoint and propagates the request. All local checkpoints taken by the participating processes for a ion collectively form a global checkpoint. The state information collected by each independent checkpointing is combined. The combination is driven by the fact that the union of consistent global checkpoints is also a consistent global checkpoint. The checkpoint thus generated is more recent than each of the checkpoints collected independently, and also more recent than that collected by [28]. Therefore, the amount of computation lost during rollback, after process failures, is minimized.

3.6 Handling Failures During Checkpointing

Since MHs are more prone to failure, there is a possibility that during the checkpointing activity, an MH fails and all processes running on it also fail. We assume that if a process fails, some processes that try to communicate with it get to know of the failure. If the failed process is not the checkpointing initiator, the simplest way to deal with failures is to use *abort* messages similar to [17, 25]. More specifically, the process detecting the failures notifies the initiator, which broadcasts *abort* messages to all process participating in the current checkpointing. Processes discard their checkpoints (tentative or mutable) and restore some variables such as $sent, old\_csn, R$, etc, on receiving the *abort* messages. If the failed process is the coordinator and the failure occurred before the process sent out *commit* or *abort* messages, on restarting after failure it broadcasts an *abort* corresponding to its checkpoint initiation. If it had failed after broadcasting a *commit* or *abort*, then it does not do anything more for that checkpoint initiation.

The above approach may not have good performance since the whole checkpointing aborts even when only one process participating in the checkpointing fails. We would like to use a more efficient approach proposed by Kim and Park [16]. In their approach, processes can commit their tentative checkpoints when none of the processes on which they depend fails. Then, the consistent recovery line is advanced for those processes that committed their checkpoints. Certainly, the initiator and other processes which depends on the failure process have to abort their checkpointing and discard their tentative (mutable) checkpoints as in [17]. In this way, the checkpoint-commitment decision can be made locally, so that the total abort of the checkpointing is avoided. That is, when a process involved in a checkpointing coordination fails, the processes not affected by the failed one can make their decisions, while the protocols in [17] aborts the whole checkpointing activity.

In mobile computing, since a wireless channel is more likely to suffer from intermittent errors, failure detection in wireless networks should be different from that in static networks. Since failure detection and failure recovery are orthogonal to our discussion, more information on how to deal with process failures can be found in [18, 22, 25].

4 Correctness Proof

In Section 3.2, R_i represents all dependency relations in the current checkpointing period. Due to the introduction of mutable checkpoints, R_i may represent the dependency relations after the last mutable checkpoint. To simplify the proof, in the following, R_i means the first parameter of subroutine *prop_cp* in our algorithm. More specifically, R_i should be $CP_i.R$ if there is a mutable checkpoint.

Theorem 1 *The algorithm creates a consistent global checkpoint.*

Proof. We prove this by contradiction. Assume that the global state of the system is inconsistent at a time instance. Then, there must be a pair of processes P_i and P_j such that at least one message m has been sent from P_j after P_j's last checkpoint and has been received by P_i before P_i's last checkpoint. Since $R_i[j]$=1 at P_i at the time of taking its checkpoint, P_i sends a checkpoint request to P_j, or a process P_k has sent the request to P_j if $MR[j].csn \geq csn_i[j]$. Thus, at least one checkpoint request is sent to P_j. If P_j runs on an MSS, the underlying network routes the request to it. If P_j runs on an MH_i, which is in MSS_p's cell, there are three possibilities when the request reaches MSS_p:

Case 1: If MH_i is still connected to MSS_p, the request is forwarded to MH_i, then to P_j.

Case 2: MH_i has moved to MSS_q (handoff). MSS_p forwards the request to MSS_q, which forwards it to MH_i, and then to P_j by the underlying routing protocol.

Case 3: MH_i is disconnected from the network. MSS_p takes a checkpoint on behalf of P_j by converting $disconnect\_checkpoint_j$ into P_j's new checkpoint (as in Section 2.2). Since P_j cannot send any message after disconnection, it must have sent m before its disconnection. Thus, the sending of m is recorded in $disconnect\_checkpoint_j$. A contradiction.

In Case 1 and Case 2, when P_j receives the request, if $req\_csn < old\_csn_j$, no matter the request comes from P_i or P_k (if the request comes from P_k: $(csn_i[j] \leq MR[j].csn) \wedge (MR[j].csn = req\_csn) \wedge (req\_csn < old\_csn_j) \Longrightarrow csn_i[j] < old\_csn_j$), P_j has already taken a checkpoint after sending m. Thus, the sending of m is recorded at P_j. If $req\_csn \geq old\_csn_j$ (the request may come from P_i or P_k), there are two possibilities:

Case 1: $own\_trigger \neq msg.trigger$. There are two possibilities for P_j to take a checkpoint:

Case 1.1: The checkpoint is taken after the sending of m. Then:

send(m) at $P_j \longrightarrow$ [1] receive(m) at P_i

receive(m) at $P_i \longrightarrow$ checkpoint taken at P_i

[1] $\longrightarrow$ is the "happened before" relation described in [20]

checkpoint taken at $P_i \longrightarrow$ *request* sent by P_i to P_j

request sent by P_i to $P_j \longrightarrow$ checkpoint taken at P_j

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

Case 1.2: The checkpoint is taken before the sending of m. As a result, P_j increase $csn_j[j]$ before it sends m to P_i, and then $m.csn > csn_i[j]$. There are two possible situations:

Case 1.2.1: P_j has finished its checkpointing process (the last checkpoint) before it sends m. Hence, P_i does not need to take a checkpoint when it receives m, and then the reception of m is not recorded in the last checkpoint of P_i.

Case 1.2.1: P_j has not finished its checkpointing process before it sends m. If P_i does not need to take a mutable checkpoint before processing m, the reception of m cannot be recorded in the last checkpoint of P_i. If P_i takes a mutable checkpoint before processing m, when P_i receives the request for this checkpoint initiation, P_i turns the mutable checkpoint into a tentative checkpoint. Certainly, the reception of m is still not recorded in the last checkpoint of P_i. □

Case 2: $own\_trigger = msg.trigger$. In this case, P_j has taken a mutable checkpoint or a tentative checkpoint. There are two possibilities:

Case 2.1: The checkpoint is taken after the sending of m. If the checkpoint is a mutable checkpoint, on receipt of the request, it is changed to a tentative checkpoint. Thus, the sending of m is recorded.

Case 2.2: The checkpoint is taken before the sending of m. Similar to Case 1.2, we get contradictions. □

Lemma 1 *Every process inherits at most one checkpoint request to take a checkpoint.*

Proof. After a process P_i inherits a checkpoint request, it changes its $own\_trigger$ to the $trigger$ attached with the request and takes a checkpoint (or make a mutable checkpoint permanent). Later, when it receives other checkpoint requests corresponding to this checkpoint initiation, we already have $own\_trigger = msg\_trigger$. Thus, P_i cannot take a mutable checkpoint; i.e., $CP_i.trigger \neq own\_trigger$. Thus, P_i cannot take a checkpoint on receipt of other requests corresponding to the same checkpoint initiation; that is, it does not inherit any request other than the first one. □

In order to prove that our non-blocking checkpointing algorithm terminates, we introduce the following notations:

- $W(request)$: the weight carried by a *request* message.
- $W(reply)$: the weight carried by a *reply* message.
- $W(P_{init})$: the weight at the initiator.
- $W(P_{other})$: the weight at a process other than the initiator.

Lemma 2 *During a checkpointing process, the following invariant holds:*

$$W(P_{init}) + \sum_{\forall request} W(request) + \sum_{\forall reply} W(reply) + \sum_{\forall P_{other}} W(P_{other}) = 1$$

Proof. When P_{init} initiates a checkpointing process, $W(P_{init}) = 1$, no weight is associated with other processes, and no *request* or *reply* messages are in transit. Hence, the invariant holds. During the checkpointing process, the initiator sends out a portion of its weight in each outgoing *request* message. Therefore,

$$\sum_{\forall request} W(request) + W(P_{init}) = 1$$

When a process P_i receives a checkpoint *request*, there are two possibilities:

Case 1: If P_i needs to take a tentative checkpoint or turn a mutable checkpoint into a tentative checkpoint, a part of the received weight is propagated to other processes in *request* messages, and the rest of the weight is sent to the initiator in a *reply*.

Case 2: If P_i does not need to take a tentative checkpoint or turn a mutable checkpoint into a tentative checkpoint, the entire received weight is sent back to the initiator in a *reply*.

Therefore, no portion of the weight in a *request* is retained by P_i. At any instant of time during the checkpointing process, *request* and *reply* messages may be in transit, and some non-initiator processes may have non-zero weights. However, no extra weight is *created* or *deleted* at any non-initiator process. Thus, the invariant holds. □

Theorem 2 *The proposed checkpointing algorithm terminates within a finite time.*

Proof. In our algorithm, a process only propagate *request* messages when it inherits a *request*. Based on Lemma 1, every process inherits at most one *request* to take a checkpoint, and then each process propagates the received *request* at most once. Since the number of processes in the system is finite, the number of *request* messages generated are finite. As message propagation delay is bounded, within a finite time after the checkpoint initiation, no new *request* messages will be generated and all such messages generated in the past have been delivered by the receivers. After this point of time, say T, the following assertion is true:

$$\sum_{\forall request} W(request) = 0 \tag{1}$$

On the receipt of a *request*, a non-initiator process immediately sends out the weight received in the *request* on the outgoing *request* messages or *reply* messages. Thus, within a finite time after T, the weight of all non-initiator processes becomes zero. As there are no more *request* messages in the system, non-initiator processes cannot acquire any weight in the future. After this point of time, say $T' > T$, the following assertion is true:

$$\sum_{\forall P_{other}} W(P_{other}) = 0 \tag{2}$$

As message propagation delay is finite, all *reply* messages will be received by the initiator within a finite time after T'. As there are no more *request* messages, no new *reply* will be generated. Hence, after time, say $T'' > T'$, the following assertion is true:

$$\sum_{\forall reply} W(reply) = 0 \tag{3}$$

Based on Lemma 2:

$$W(P_{init}) + \sum_{\forall request} W(request) + \sum_{\forall reply} W(reply) + \sum_{\forall P_{other}} W(P_{other}) = 1$$

After time T'', since $T'' > T' > T$, assertions (1), (2), and (3) are all true. Thus, $W(P_{init}) = 1$. At this point, the initiator sends *commit* messages to the processes that took checkpoints. A non-initiator process receives the *commit* message within a finite time. Therefore, the checkpointing algorithm terminates within a finite time. □

We now show that the number of processes that take new tentative (permanent) checkpoints during the execution of our algorithm is minimal. Based on Lemma 1, a process takes at most one checkpoint corresponding to a checkpointing process. Let $\mathcal{P} = \{P_0, P_1, \cdots, P_k\}$ be the set of processes that take new checkpoints during the execution of our algorithm, where P_0 is the initiator. Let $\mathcal{C}(\mathcal{P}) = \{C(P_0), C(P_1), \cdots, C(P_k)\}$ be the new checkpoints taken by processes in $\mathcal{P}$.

When a process receives a checkpoint request, it asks all processes on which it depends to take checkpoints. The process receiving the request should take a checkpoint as soon as possible, since the longer it waits, the more processes will have dependency relation with it, and then the more processes need to take checkpoints. If the initiator knows all processes on which it depends, it can send checkpoint requests to them at once and then save the time of tracing the dependency tree. Some techniques [6, 25] exist to approximate this approach. However, it increases run time overhead since extra information has to be appended with the computation messages. Since the message delay is far less than the time between two checkpoint intervals, we do not consider the extra checkpoints resulting from the checkpoint request delay. Our algorithm can also use the techniques in [6, 25], but, as we discussed, that increases run time overhead and it is not valuable.

We define an alternate set of checkpoints: $\mathcal{C}'(\mathcal{P}) = \{C'(P_0), C'(P_1), \cdots, C'(P_k)\}$ where $C'(P_0) = C(P_0)$, and $C'(P_i)$ $(1 \leq i \leq k)$ is either $C(P_i)$ or the checkpoint P_i had taken before executing our algorithm. If $C'(P_i)$ is a new checkpoint, as we discussed, it should be taken as soon as possible, and then it is equal to $C(P_i)$ without considering the checkpoint request delay.

Theorem 3 *$\mathcal{C}'(\mathcal{P})$ is consistent if and only if $\mathcal{C}'(\mathcal{P}) = \mathcal{C}(\mathcal{P})$.*

Proof. The *if* part directly comes from Theorem 1. We now prove the *only if* part. The execution of our algorithm imposes a "P_i inherits a request from P_j" (defined in Section 3.2) relation on the set of processes. Since this relation is non-circular (based on Lemma 1) and there is only one initiator, it can be represented as a tree T: the root of T is the initiator, and P_j is a child of P_i if and only if P_j inherits a request from P_i. If $P_j \in T$, it must take a new checkpoint during the execution of the algorithm; hence $P_j \in \mathcal{P}$. If $P_j \in \mathcal{P}$, either P_j is the initiator or it inherits a request; hence $P_j \in T$. Therefore, $P_j \in T$ if and only if $P_j \in \mathcal{P}$.

Our proof is by contradiction. Suppose $\mathcal{C}'(\mathcal{P}) \neq \mathcal{C}(\mathcal{P})$ and $\mathcal{C}'(\mathcal{P})$ is consistent. Let $P_j \in \mathcal{P}$ such that $C'(P_j) \neq C(P_j)$. Note that $P_j \neq P_0$, and there exists a path from P_0 to P_k in T. Since $C'(P_0) = C(P_0)$, there is an edge (P_i, P_j) on this path such that $C'(P_i) = C(P_i) \wedge C'(P_j) \neq C(P_j)$. Let m be the last message P_i receives from P_j. Since P_j

inherits P_i's request, we have $req\_csn \geq old\_csn_j$ (req_csn is appended with the request), and the receipt of m is recorded in $C(P_i)$ (or $C'(P_i)$). Also, the sending of m is recorded in $C(P_j)$. If $C(P_j) \neq C'(P_j)$, $C'(P_j)$ is the checkpoint P_j had before executing the algorithm, and then the sending of m is not recorded in $C'(P_j)$. Thus, $\mathcal{C}'(\mathcal{P})$ is not a consistent set of checkpoints. A contradiction. □

5 A Performance Evaluation

A mutable checkpoint is *redundant* if it will not be turned into a tentative checkpoint. In this section, to evaluate the performance of our algorithm, we first use simulations to measure the number of redundant mutable checkpoints taken during a *checkpointing period*, which is the duration of a checkpointing process, from the initiation to the termination. Then, we compare our algorithm with other algorithms in the literature.

5.1 Simulation Model

A system with N MHs connected through a wireless LAN is simulated. Each MH has one process running on it and N is equal to 16. The wireless LAN has a bandwidth of $2Mbps$, which follows the IEEE 802.11 standard [9]. The length of each computation message is $1KB$; Thus, the transmission delay of each computation message is $8 * 1/2 = 4ms$. The length of each system message is 50Bytes. Thus, the transmission delay of each system message is $0.05 * 8/2 = 0.2ms$. The size of a checkpoint is $1MB$ [11]. We can use incremental checkpointing [11] to reduce the amount of data that must be written on stable storage; that is, only the pages of the address space that have been modified since the previous checkpoint are transferred to the MSS. As a result, we assume that only $500KB$ are transmitted over the wireless link in order to take a tentative checkpoint, which needs $0.5 * 8/2 = 2s$ (disk access time is not counted). In today's technology, Pentium $266MHz$ laptops with $64MB$ memory are available and will become popular soon. Thus, we assume mutable checkpoints are saved in the main memory. Since processor speed is much faster, the main memory is the bottleneck. For typical Pentium technology, 64bit wide memory bus with $83MHz$ bus speed is used. Thus, it needs about $\frac{1*2}{83*8} = 3ms$ to save a mutable checkpoint. (If memory block copy is supported, the time can be further reduced.) A checkpoint is scheduled at each process with an interval of 900 seconds. If a process takes a checkpoint before its scheduled checkpoint time, the next checkpoint will be scheduled $900s$ after that time. For simplicity, concurrent initiation, handoff and failures are not considered.

Each process sends out computation messages with the time interval following an exponential distribution. The message receiving pattern is considered in two computation environments: *point-to-point communication* and *group communication*. In the point-to-point communication, the destination of each message is uniformly distributed among all processes. In the group communication, processes are arranged into four groups and each group has a group leader. For intra-group communication, the destination of each message is a uniformly distributed random variable among all group members. Only group leaders can have inter-group communication, where the destination of each message is a uniformly distributed random variable among all group leaders.

5.2 Simulation Results

Since saving checkpoints takes a long time, in order not to block the process's execution, we use *pre-copying* [15, 31]; that is, the pages are copied to a separate area in the main memory and are then written from there to the stable storage. This is similar to saving a mutable checkpoint first and then turning it into a tentative checkpoint. Thus, we do not measure the number of mutable checkpoints that will be turned into tentative checkpoints. We measure the number of tentative checkpoints and the number of redundant mutable checkpoints for each checkpoint initiation under various message sending rate. The mean value of a measured parameter is obtained by collecting a large number of samples such that the confidence interval is reasonably small. In most cases, the 95% confidence interval for the measured data is less than 10% of the sample mean.

5.2.1 Point-to-point communication

As shown in Figure 5, the number of tentative checkpoints for each checkpoint initiation increases as the message sending rate increases. Since the message receiving event is uniformly distributed, a process is more likely to receive a message from other processes when the message sending rate increases. Thus, it is more likely to have a dependency relationship with the initiator and thus is more likely to take a tentative checkpoint.

In Figure 5, when the message sending rate increases, the number of redundant mutable checkpoints for each checkpoint initiation increases at first and then decreases, and it is always less than 4% of the number of tentative checkpoints. This can be explained as follows. A process takes a mutable checkpoint only when it receives a computation message before it receives the checkpoint request during the checkpointing period. It takes a tentative checkpoint if it has received messages that created dependency relationship with the initiator during the checkpoint interval. Since the checkpointing period (at most $2 * 16 = 32s$ long) is much less than the checkpoint interval (900s), in general, a process takes much fewer redundant mutable checkpoints than tentative checkpoints. If the message sending rate is low, processes have low probability to send messages, and they have low probability to receive messages during the checkpointing period. Thus, they have low probability to take mutable checkpoints. If the message sending rate is high, it is more likely for a process to receive a message and take a mutable checkpoint during the checkpointing period. The mutable checkpoint is also more likely to be turned into a tentative checkpoint, and then it is not a redundant mutable checkpoint. Also, according to our algorithm, the initiator quickly propagates the checkpoint request; thus, a process is less likely to receive a computation message before the checkpoint request during the checkpoint period, and then it is less likely to take a mutable checkpoint.

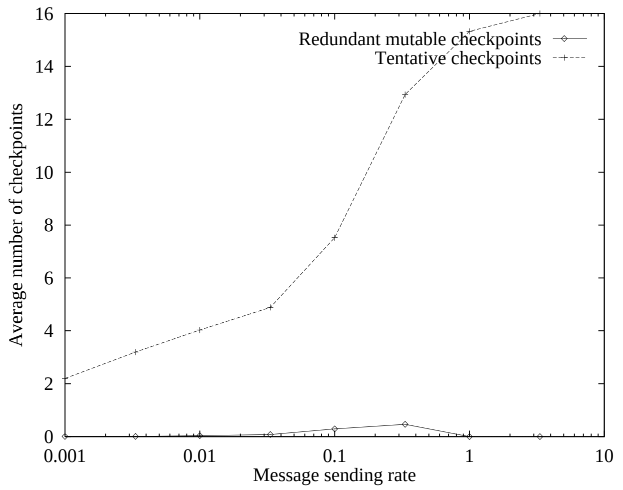

Figure 5: The number of checkpoints in point-to-point communication environment

5.2.2 Group communication

Figure 6 shows the number of checkpoints in a group communication environment. Besides changing the intra-group message sending rate, a group leader also changes its inter-group message sending rate. In the left graph of Figure 6, for a group leader, the intra-group message sending rate is 1000 times more than the inter-group message sending rate; while in the right graph of Figure 6, the intra-group message sending rate is 10000 times more. As can be seen, with group communication, the number of tentative checkpoints and redundant mutable checkpoints in the right graph is less than that in the left graph, and they are smaller than that in the point-to-point communication. In a group communication, when a process initiates a checkpointing process, processes in other groups have low probability of receiving messages from any process in the initiator's group. Thus, they are less likely to have dependency relationship with the initiator; that is, they have low probability of taking tentative checkpoints or redundant mutable checkpoints.

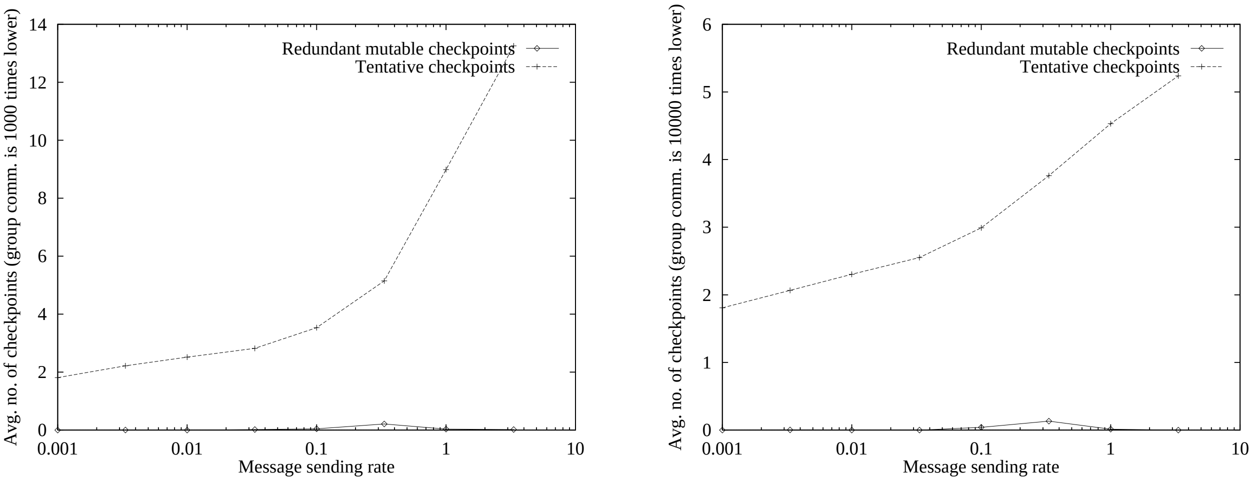

Figure 6: The number of checkpoints in a group communication environment

5.3 Comparison With Other Algorithms

The following notations are used to compare our algorithm with other algorithms.

Notations

C_{air}: cost of sending a message from one process to another process.

C_{broad}: cost of broadcasting a message to all processes.

T_{disk}: delay incurred in saving a checkpoint on stable storage in an MSS.

T_{data}: delay incurred in transferring a checkpoint from an MH to its MSS.

T_{msg}: delay incurred by system messages during a checkpointing process.

T_{ch}: the time of a checkpointing period. $T_{ch} = T_{msg} + T_{data} + T_{disk}$.

N_{min}, N, N_{muta}, N_{dep}: N_{min} is the number of processes that need to take checkpoints using the Koo-Toueg algorithm [17]. N is the total number of processes in the system. N_{muta} is the number of redundant mutable checkpoints during a checkpointing process. N_{dep}

is the average number of processes on which a process depends. Note that $1 \leq N_{dep} \leq N - 1$.

We use five parameters to evaluate the performance of a checkpointing algorithm: the number of tentative checkpoints required during a checkpointing process, the blocking time (in the worst case), the system message overhead, whether the algorithm is distributed or not, and the *output commit* delay, which is the delay incurred before the system commits to the *outside world*. The outside world consists of everything with which processes can communicate that does not participate in the system's rollback-recovery, such as the user's workstation display, or even the file system if no special support is available for rolling back the contents of files. Messages sent to the outside world must be delayed until the system can guarantee that the message will never be "unsent" as a result of processes rolling back to recover from any possible future failure. If the sender is forced to roll back to a state before the message was sent, recovery of a consistent system state may be impossible, since the outside world cannot in general be rolled back. Once the system can meet this guarantee, the message may be committed by releasing it to the outside world. Generally, if a process needs output commit, it initiates a checkpointing process. Thus, the output commit delay equals the duration of the checkpointing process.

Performance of our algorithm

It is easy to see that our algorithm is distributed and the blocking time is 0.

The number of tentative checkpoints: Based on the result of Theorem 3, our algorithm forces only a minimum number of processes to save checkpoints on stable storage.

The output commit delay: From the simulation results, the number of redundant mutable checkpoints is less than 4% of the number of tentative checkpoints. Based on our simulation parameters, the delay of taking a tentative checkpoint is almost 1000 times longer than that of taking a mutable checkpoint. Thus, the output commit delay of our algorithm is approximately $N_{min} * T_{ch}$. Note that for some applications, the checkpoint size is pretty small and the wireless network may have high bandwidth in the future, but at that time, the memory bus bandwidth also becomes larger. Moreover, at that time, the disk access delay, which is difficult to reduce, may dominate the checkpointing time. Thus, the delay of taking a tentative checkpoint is still significantly longer than that of taking a mutable checkpoint.

The system message overhead: In the first phase, a process taking a tentative checkpoint needs two system messages: *request* and *reply*. A process may receive more than one request for the same checkpoint initiation from different processes. However, we have used some techniques to reduce this kind of situation to occur. Thus, the system message overhead is approximately $2 * N_{min} * C_{air}$ in the first phase. In the second phase, we hope to get the advantages of the update approach and the broadcast approach by system tuning. Thus, the system message overhead is approximately $min(N_{min} * C_{air}, C_{broad})$ in the second phase.

Comparison to other algorithms

Table 1 compares our algorithm with two representative approaches for coordinated checkpointing. The Koo-Toueg algorithm [17] has the lowest overhead (based on our five parameters) among the blocking algorithms [2, 8, 10, 16, 17, 21, 26] which try to minimize the number of synchronization messages and the number of checkpoints. The algorithm in [11] has the lowest overhead (based on our five parameters) among the non-blocking algorithms [7, 11, 19, 27]. We do not compare our algorithm with the Prakash-Singhal algorithm since it may result in inconsistencies, and there is no easy solution to fix it without increasing overhead.

As shown in Table 1, when compared to the Koo-Toueg algorithm, our algorithm reduces the message overhead from $3 * N_{min} * N_{dep} * C_{air}$ $(1 \leq N_{dep} \leq N - 1)$ to $2 * N_{min} * C_{air} + min(N_{min} * C_{air}, C_{broad})$. When $N_{min} = N$, the message reduction can be from $O(N^2)$ to $O(N)$. Our algorithm reduces the blocking time from $N_{min} * T_{ch}$ to 0. In the worst case, $N_{min} = N$. Consider our simulation parameters: $N = 16$ and $T_{ch} = 2s$, the blocking time will be $32s$; i.e., all processes cannot do anything for half a minute in the Koo-Toueg algorithm, which significantly reduces system performance. Compared to [11], our algorithm forces only a minimum number of processes to take their checkpoints on stable storage. Note that there may be many applications running in the system: some of them have higher reliability requirement and others do not. In a heterogeneous environment, some MHs may be more prone to failures than others. Moreover, different processes may run at their own speed and they may only communicate with a group of processes. As a result, some processes may need to take checkpoints more frequently than others. However, the algorithm in [11] forces all processes in the system to take checkpoints for each checkpoint initiation. Thus, our algorithm significantly reduces the message overhead and checkpointing overhead compared to [11]. Furthermore, in case of output commit, our algorithm has much shorter delay compared to [11] since our algorithm requires fewer processes to take checkpoints before committing to the outside world. Seems like that our algorithm needs more system

Algorithm	Checkpoints	Blocking time	Output commit	Messages	Distributed
Koo-Toueg [17]	N_{min}	$N_{min} * T_{ch} = N_{min} * (T_{msg} + T_{data} + T_{disk})$	$N_{min} * T_{ch}$	$3 * N_{min} * N_{dep} * C_{air}$	Yes
Elnozahy [11]	N	0	$N * T_{ch}$	$2 * C_{broad} + N * C_{air}$	No
Our algorithm	N_{min}	0	$(N_{min} + N_{muta}) * T_{ch} \approx N_{min} * T_{ch}$	$\approx 2 * N_{min} * C_{air} + min(N_{min} * C_{air}, C_{broad})$	Yes

Table 1: **A comparison of system performance**

messages than [11]. However, the algorithm in [11] is a centralized algorithm and there is no easy way to make it distributed without significantly increasing message overhead. Since some processes may be in the doze mode, broadcast may waste their energy and processor power. More importantly, the system message is relatively small, and then the overhead of system messages is much smaller compared to the overhead of saving checkpoints on stable storage.

6 Related Work

The first coordinated checkpointing algorithm was presented in [2]. However, it assumes that all communications between processes are atomic, which is too restrictive. The Koo-Toueg algorithm [17] relaxes this assumption. In this algorithm, only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints. Thus, it reduces the number of synchronization messages and the number of checkpoints. Later, Leu and Bhargava [21] presented an algorithm, which is resilient to multiple process failures, and does not assume that the channel is $FIFO$, which is necessary in [17]. However, these two algorithms [17, 21] assume a complex scheme (such as slide window) to deal with the message loss problem and do not consider lost messages in checkpointing and recovery. Deng and Park [10] proposed an algorithm to address both orphan message and lost messages.

In Koo and Toueg's algorithm [17], if any of the involved processes is not able to or not willing to take a checkpoint, the entire checkpointing process is aborted. Kim and Park [16] proposed an improved scheme that allows the new checkpoints in some subtrees to be committed while the others are aborted.

To further reduce the system messages needed to synchronize the checkpointing, loosely synchronous clocks [8, 26] are used. More specifically, loosely-synchronized checkpoint clocks can trigger the local checkpointing actions of all participating processes at approximately the same time without the need of broadcasting the checkpoint request by the initiator. However, a process taking a checkpoint needs to wait for a period that equals to the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process in the system.

All the above coordinated checkpointing algorithms [2, 8, 10, 16, 17, 21, 26] require processes to be blocked during checkpointing. Checkpointing includes the time to trace the dependency tree and to save the state of processes on stable storage, which may be long. Therefore, blocking algorithms may dramatically reduce the performance of the system [4, 11].

The Chandy-Lamport algorithm [7] is the earliest non-blocking algorithm for coordinated checkpointing. However, in their algorithm, system messages (markers) are sent along all channels in the network during checkpointing. This leads to a message complexity of $O(N^2)$. Moreover, it requires all processes to take checkpoints and the channel must be $FIFO$. To relax the $FIFO$ assumption, Lai and Yang [19] proposed another algorithm. In their algorithm, when a process takes a checkpoint, it piggybacks a checkpoint request (a flag) to the messages it sends out from each channel. The receiver checks the piggybacked message flag to see if there is a need to take a checkpoint before processing the message. If so, it takes a checkpoint before processing the message to avoid an inconsistency. To record the channel information, each process needs to maintain the entire message history on each channel as part of the local checkpoint. Thus, the space requirements of the algorithm may be large. Moreover, it requires all processes to take checkpoints, even though many of them are unnecessary.

The Elnozahy-Johnson-Zwaenepoel algorithm [11] uses the checkpoint sequence number to identify orphan messages, thus avoiding the need for processes to be blocked during checkpointing. However, this approach requires the initiator to communicate with all processes in the computation. The algorithm proposed by Silva and Silva [27] uses a similar idea as [11] except that the processes which did not communicate with others during the previous checkpoint interval do not need to take new checkpoints. Both algorithms [11, 27] assume that a distinguished initiator decides when to take a checkpoint. Therefore, they suffer from the disadvantages of centralized algorithms, such as one-site failure, traffic bottle-neck, etc. Moreover, their algorithms require almost all processes to take checkpoints, even though many of them are unnecessary. If they are modified to permit more processes to initiate checkpointing, which makes them distributed, the new algorithm suffers from another problem; in order to keep the checkpoint sequence number updated, any time a process takes a checkpoint, it has to notify all processes in the system. If each process can initiate a checkpointing process, the network would be flooded with control messages and processes might waste their time taking unnecessary checkpoints.

All the above algorithms follow two approaches to reduce the overhead associated with coordinated checkpointing algorithms: one is to minimize the number of synchronization messages and the number of checkpoints [2, 8, 10, 16, 17, 21, 26]; the other is to make checkpointing non-blocking [7, 11, 19, 27]. These two approaches were orthogonal in previous years until the Prakash-Singhal algorithm [25] combined them. However, their algorithm has some problems and may result in an inconsistency [5].

Acharya and Badrinath [1] were the first to present an checkpointing algorithm for mobile computing system. In their uncoordinated checkpointing algorithm, an MH takes a local checkpoint whenever a message reception is preceded by a message sent at that MH. If the send and receive of messages are interleaved, the number of local checkpoints will be equal to half of the number of computation messages, which may degrade the system performance.

For other uncoordinated checkpointing algorithms, as described in [3, 29], every process may accumulate multiple local checkpoints and logs on stable storage during normal operation. A checkpoint can be discarded if it is determined that it will no longer be needed for recovery. For this purpose, processes have to periodically broadcast the status of their logs on stable storage. The number of local checkpoints depends on the frequency with which such checkpoints are taken, and is an algorithm tuning parameter. An uncoordinated checkpointing approach is not suitable for mobile computing for a number of reasons. If the frequency of local checkpointing is high, each process will have multiple checkpoints requiring a large amount of stable storage, which also introduces a lot of communication overhead in mobile computing system. The stable storage and communication overheads can be reduced by taking local checkpoints less frequently. However, this will increase the recovery time as greater rollback and reply will be needed. Even though some algorithms [22, 32] were proposed to reduce the number of checkpoints to be saved on stable storage, to ensure correctness, a process still needs to keep much more checkpoints in uncoordinated checkpointing algorithms than that in coordinated checkpointing algorithms. In the coordinated checkpointing algorithm presented in this paper, most of the time, each process needs to store only one permanent checkpoint on stable storage, and at most two checkpoints: a permanent and a tentative (or mutable) checkpoint only for the duration of the checkpointing. Generally speaking, uncoordinated checkpointing approaches suffer from the complexities of finding a consistent recovery line after the failure, the susceptibility to the domino effect, the high stable storage overhead of saving multiple checkpoints of each process, and the overhead of garbage collection. Thus, our coordinated checkpointing algorithm has many advantages over uncoordinated checkpointing algorithms.

7 Conclusions

Mobile computing is a rapidly emerging trend in distributed computing. A mobile computing system consists of mobile hosts (MHs) and mobile support stations (MSSs), connected by a communication network. The mobility of MHs in mobile computing systems generates many new constraints, such as handoffs, lack of stable storage, low communication bandwidth of a wireless channel and energy conservation, which make traditional checkpointing algorithm unsuitable. These new constraints require that the checkpointing algorithm should be non-blocking and only forces minimum number of processes to take checkpoints. However, according to our previous result [5], there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints. In order to design an efficient checkpointing algorithm for mobile computing systems, we introduced a new concept called "mutable checkpoint", which is neither a tentative checkpoint nor a permanent checkpoint, but it can be turned into a tentative checkpoint. Mutable checkpoints can be saved anywhere; e.g., the main memory or local disk of MHs. In this way, taking a mutable checkpoint avoids the overhead of transferring large amount of data to the stable storage at MSSs over the wireless network. We also presented techniques to minimize the number of mutable checkpoints. Simulation results show that the overhead of taking mutable checkpoints is negligible. Based on mutable checkpoints, our non-blocking algorithm avoids the avalanche effect and forces only a minimum number of processes to take their checkpoints on stable storage.

References

[1] Arup Acharya and B.R. Badrinath. "Checkpointing Distributed Applications on Mobil Com-

puters". *the Third International Conference on Parallel and Distributed Information Systems*, September 1994.

[2] G. Barigazzi and L. Strigini. "Application-Transparent Setting of Recovery Points". *Digest of Papers FTCS-13*, pages 48–55, 1983.

[3] B. Bhargava and S. Lian. "Independent Checkpointing and Concurrent Rollback for Recovery inDistributed Systems". *Proceedings of the* 7^{th} *IEEE symposium on Reliable Distributed System*, pages 3–12, October 1988.

[4] B. Bhargava, S.R. Lian, and P.J. Leu. "Experimental Evaluation of Concurrent Checkpointing and Rollback-Recovery Algorithms". *Proceedings of the International Conference on Data Engineering*, pages 182–189, 1990.

[5] G. Cao and M. Singhal. "On the Impossiblity of Min-Process Non-Blocking Checkpointing and An Efficient Checkpointing Algorithm for Mobile Computing Systems". *The* 27^{th} *International Conference on Parallel Processing*, August 1998.

[6] G. Cao and Mukesh Singhal. "Low-Cost Checkpointing with Mutable Checkpoints in Mobile Computing Systems". *The* 18^{th} *International Conference on Distributed Computing Systems*, pages 464–471, May 1998.

[7] K.M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems*, February 1985.

[8] F. Cristian and F. Jahanian. "A timestamp-based Checkpointing Protocol for long-lived Distributed Computations". *Proc. of IEEE Symp. Reliable Distributed Syst.*, pages 12–20, 1991.

[9] B. Crow, I. Widjaja, J. Kim, and P. Sakai. "IEEE 802.11 Wireless Local Area Networks". *IEEE Communications Magazine*, pages 116–126, September 1997.

[10] Y. Deng and E.K. Park. "Checkpointing and Rollback-Recovery Algorithms in Distributed Systems". *Journal of Systems and Software*, pages 59–71, April 1994.

[11] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. "The Performance of Consistent Checkpointing". *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 86–95, October 1992.

[12] G.H. Forman and J. Zahorjan. "The Challenges of Mobile Computing". *IEEE Computer*, pages 38–47, April 1994.

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

[14] J. Ioannidis, D. Duchamp, and G.Q. Maguire. "Ip-based Protocols for Mobile Internetworking". *Pro. of ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, pages 235–245, September 1991.

[15] D. Johnson. "Distributed System Fault Tolerance Using Message Logging and Checkpointing". *PhD Thesis, Rice University*, December 1989.

[16] J.L. Kim and T. Park. "An Efficient Protocol For Checkpointing Recovery in Distributed Systems". *IEEE Transactions on Parallel and Distributed Systems*, pages 955–960, August 1993.

[17] R. Koo and S. Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems". *IEEE Transactions on Software Engineering*, pages 23–31, January 1987.

[18] P. Krishna, N.H. Vaidya, and D.K. Pradhan. "Recovery in Distributed Mobile Environments". *IEEE Workshop on Advances in Parallel and Distributed System*, October 1993.

[19] T.H. Lai and T.H. Yang. "On Distributed Snapshots". *Information Processing Letters*, pages 153–158, May 1987.

[20] L. Lamport. "Time, Clocks and Ordering of Events in Distributed Systems". *Communication of the ACM*, July 1978.

[21] P.Y. Leu and B. Bhargava. "Concurrent Robust Checkpointing and Recovery in Distributed Systems". *Pro. 4th IEEE Int. Conf. on Data Eng.*, pages 154–163, 1988.

[22] D. Manivannan and Mukesh Singhal. "A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing". *Proceedings of the* 16^{th} *International Conference on Distributed Computing Systems*, pages 100–107, May 1996.

[23] R.H.B. Netzer and Jian Xu. "Necessary and Sufficient Conditions for Consistent Global Snapshots". *IEEE Transactions on Parallel and Distributed System*, pages 165–169, February 1995.

[24] Ravi Prakash and Mukesh Singhal. "Maximal Global Snapshot with Concurrent Initiators". *Proceedings of the Sixth IEEE symposium on Parallel and Distributed Processing*, pages 344–351, October 1994.

[25] Ravi Prakash and Mukesh Singhal. "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems". *IEEE Transactions on Parallel and Distributed System*, pages 1035–1048, October 1996.

[26] P. Ramanathan and K.G. Shin. "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System". *IEEE Transactions on Software Engineering*, pages 571–583, June 1993.

[27] L.M. Silva and J.G. Silva. "Global Checkpointing for Distributed Programs". *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, October 1992.

[28] M. Spezialetti and P. Kearns. "Efficient Distributed Snapshots". *Proceedings of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[29] R.E. Strom and S.A. Yemini. "Optimistic Recovery In Distributed Systems". *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[30] F. Teraoka, Y. Yokote, and M. Tokoro. "A Network Architecture Providing Host Migration Transparency". *Proc. of ACM SIGCOMM'91*, September 1991.

[31] M. Theimer, K. Lantz, and D. Cheriton. "Preemptable Remote Execution Facilities in the V-system". *The 10th ACM Symposium on Operating Systems Principles*, pages 1035–1048, December 1985.

[32] Y. Wang and W.K. Fuchs. "lazy Checkpoint Coordination for Bounding Rollback Propagation". *The* 12^{th} *Symposium on Reliable Distributed Systems*, pages 78–85, October 1993.