© The Author 2006. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
Advance Access published on February 17, 2006
doi:10.1093/comjnl/bxk004

Design and Evaluation of a Low-Latency Checkpointing Scheme for Mobile Computing Systems

GUOHUI LI[1] AND LIHCHYUN SHU[2,*]

[1]*School of Computer Science and Technology, Huazhong University of Science and Technology, P. R. China*
[2]*Department of Accountancy, National Cheng Kung University, Taiwan 701, ROC*
Corresponding author: shulc@mail.ncku.edu.tw

Fault-tolerant mobile computing systems have different requirements and restrictions, not taken into account by conventional distributed systems. This paper presents a coordinated checkpointing scheme which reduces the delay involved in a global checkpointing process for mobile systems. A piggyback technique is used to track and record the checkpoint dependency information among processes during normal message transmission. During checkpointing, a concurrent checkpointing technique is designed to use the pre-recorded process dependency information to minimize process blocking time by sending checkpoint requests to dependent processes at once, hence saving the time to trace the dependency tree. We show that our checkpoint algorithm forces a minimum number of processes to take checkpoints, which is an important property for checkpointing mobile applications. Via probability-based analysis, we show that our scheme can significantly reduce the latency associated with checkpoint request propagation, compared with traditional coordinated checkpointing approaches. Experimental results indicate that we have <2% overhead in transmitting piggybacked information during normal runtime. However, we can achieve up to a 60% reduction in checkpoint latency time.

Keywords: Fault tolerance, mobile computing systems, rollback recovery, causal dependency, coordinated checkpointing

Received 1 September 2005; revised 5 January 2006

1. INTRODUCTION

Although checkpointing techniques for distributed computing systems have been extensively studied in the last two decades, most of the previous works assumed that systems were built on wired networks. Lately, we have witnessed tremendous research interests in mobile computing systems. Research on the problem of devising efficient checkpointing algorithms for mobile computing systems has started to emerge [1, 2, 3]. It is well recognized that fault-tolerant solutions developed for conventional distributed systems are not appropriate in the mobile environments due to the characteristics of mobile networks and devices. For example, because mobile appliances are vulnerable to physical damages, mobile clients' storage is considered to be unreliable. Hence, the checkpoint state of a process will need to be transferred to stable storages, e.g. those on the mobile support stations (MSSs) [4]. Since wireless network has limited bandwidth and mobile clients have limited computation power, it is most desirable that a coordinated checkpoint algorithm forces a minimum number of processes to take checkpoints.

In the mobile environment, mobile hosts (MHs) relocate from time to time. In order to reduce the overhead associated with locating a mobile client, it becomes critical to reduce the checkpoint latency from the time a process initiates a checkpoint request to the time the global checkpointing process completes. This requirement makes sequential coordinated schemes, e.g. Koo and Toueg's algorithm [5], inappropriate. With such an algorithm, a checkpoint initiating process notifies other processes which are directly checkpoint dependent on it. These other processes then in turn notify processes which are directly dependent on these other processes. This global checkpointing operation essentially walks through a checkpoint dependency tree, from the root to the leaves, one level at a time until all relevant processes have been informed. Furthermore, the acknowledgement procedure is also sequential from the processes at the leaf levels through their ancestors, one level at a time, until the root (initiating) process receives all responses.

In the mobile environment, it is also desirable to minimize the number of synchronization messages that must be transmitted when checkpointing is in progress due to mobile environment's low communication bandwidth constraint and energy conservation requirement. However, this goal is not easy to achieve without incurring additional overhead. In particular, it is sometimes a trade-off between reducing the number of synchronization messages and reducing the amount of information carried on each synchronization message.

In this paper, we propose a concurrent checkpointing technique which aims to reduce the checkpoint latency in the mobile environment. The central idea is to identify checkpoint-dependencies during regular (non-checkpointing) operation, by tagging all computational messages with dependency information. Having this dependency information at hand, a checkpoint initiator can simultaneously inform all of its dependents to take their checkpoints. In addition, we propose an acknowledgement scheme so that each dependent process can directly reply to the initiator.

Besides reducing checkpoint latency, our protocol is shown to force a minimum number of processes to take checkpoints, which is a very desirable property for mobile applications. We have evaluated our scheme both analytically and experimentally. Results show that our scheme can significantly reduce the checkpoint latency, compared with traditional coordinated checkpointing approaches. The overhead incurred in transmitting piggybacked information is, however, rather small.

We organize the remainder of the paper as follows. Related work is provided in Section 2. Section 3 introduces the notion of checkpoint dependency. We describe the main problem associated with existing coordinated checkpointing techniques, which motivates our work. In Section 4, we describe our idea of concurrent checkpointing approach as a way to efficiently enforce global checkpoint consistency. We also describe a technique to cope with tardy messages that could arise during message transmission. In Section 5, we discuss correctness properties and expected performance of our concurrent checkpoint algorithm. Section 6 presents the performance evaluation of our approach via simulation experiments. Section 7 concludes the paper.

2. RELATED WORK

Rollback recovery in distributed systems is an extensively researched area, as evidenced in a recent *ACM Computing Surveys* article [6]. Additional comparisons from the perspectives of failure-free operation cost and failure recovery cost on various recovery schemes, including checkpointing and logging, can be found in [2]. It is well recognized that fault-tolerant solutions developed for conventional distributed systems are not appropriate in the mobile environments due to the characteristics of mobile networks and devices. To our knowledge, the earliest work on checkpointing mobile applications is due to Acharya and Badrinath [4]. Their uncoordinated checkpointing approach requires a MH to take a local checkpoint whenever a message reception is preceded by a message sending event by the MH. It is not hard to see that this approach could incur significant checkpoint overhead if the message sending and reception events are interleaved. Pradhan *et al*. [7] discussed the limitations of the mobile environment, and its effects on recovery protocols. They found that the performance of a recovery scheme mainly depends on three parameters: the failure rate of the MHs, the mobility of the hosts and the wireless bandwidth.

Cao and Singhal [9] proposed a blocking algorithm for mobile systems which exploits the computational capabilities of MSSs. When an MH initiates a checkpoint operation, the checkpoint request is first sent to its current MSS, say MSSq, which then becomes the proxy of the checkpoint operation. MSSq will broadcast request messages to all other MSSs to retrieve corresponding data structures and perform a matrix multiplication to determine which processes should take checkpoints. The problem associated with this approach is that when the number of MHs and processes in the system becomes large, the matrix can be very large and the computation cost can be very expensive. In other words, this approach may not scale well.

Cao and Singhal [1] recently proposed an algorithm which leverages the advantages of non-blocking schemes, and at the same time reduces the number of possibly redundant checkpoints (an unavoidable property of non-blocking algorithms proved in [9]), called mutable checkpoints. Mutable checkpoints can be saved on MHs, hence avoiding the overhead of transferring large amount of data to the stable storage at MSSs. While Cao and Singhal [1] showed that their approach forces a minimum number of processes to take permanent checkpoints, it has been observed by Kumar *et al*. [10] that many of the mutable checkpoints may be useless, leading to increased checkpoint overhead to MHs.

Manabe [11] proposed a checkpointing approach which specifically addresses the handoff situation in wireless networks. When a handoff occurs, the process experiencing the handoff takes a checkpoint when the process reconnects. While taking handoff checkpoints is due to higher failure probability at handoff time, this design implies significant checkpoint overhead should the mobile device on which the process executes keeps moving from cell to cell. Chen *et al*. [8] addressed failure recovery of client–server applications in the mobile environment. They derived closed-form expressions for the recovery time probability distribution with respect to different handoff strategies. This allows a designer to select an appropriate handoff strategy for a failure recovery scheme under different system characteristics.

To avoid coordination messages in coordinated checkpointing protocols, time-based protocols have been proposed for mobile systems [3, 12]. These protocols operate by using synchronized clocks. The protocol proposed by Lin *et al*. [3] tries to reduce the number of checkpoints compared with the traditional time-based protocols. Their idea is in some sense similar to Cao and Singhal's mutable checkpoints: a soft checkpoint is taken by every process during a checkpointing process. A soft checkpoint is discarded if the owning process is found to be irrelevant; otherwise, it is saved in the stable storage.

A rollback recovery protocol based on optimistic message logging, rather than checkpointing, was recently proposed for mobile systems in [2]. Compared with the checkpointing schemes, message logging schemes require higher stable storage access overhead. However, the recomputation cost in the optimistic logging schemes is much smaller and asynchronous recovery is possible. Hence, such schemes should be a good choice for mobile applications when speedy recovery is necessary. Yao *et al*. [13] proposed a receiver-based pessimistic message logging protocol. Similar to [2], this protocol permits quick asynchronous recovery for processes running on MHs. However, its storage access overhead can be even higher than that for optimistic logging.

3. SYSTEM MODEL AND BACKGROUND

In this section, we first describe our system model and the notion of checkpoint dependency in Section 3.1. In Section 3.2, we describe the main problem associated with existing coordinated checkpointing techniques, which motivates our work.

3.1 Notations and checkpoint dependency

We consider a set of processes running concurrently on fail-stop MHs or mobile support stations MSSs in the network. Message passing is the only way for processes to communicate with each other. Each process executes at its own pace and messages are exchanged through reliable communication channels whose transmission delays are finite but arbitrary.

As a result of inter-process communications, the state of a process may depend directly or indirectly on other processes. If a process P has to roll back due to a failure, the processes that directly or indirectly depend on P's state must also roll back. During recovery, it is important that the system is recovered to a global consistent state and then continues its operations from this consistent state. A global state consists of the local states of all the processes, with each local state coming from an active process. A global state is *consistent* if it contains no orphan message, i.e. a message whose received event is recorded in the local state of the destination process, but its send event is lost in the local state of the source process [14].

We assume each checkpoint is associated with a unique sequence number. The sequence number of a process P_i increases monotonically. The checkpoint with sequence number m of a process P_i is denoted as $C_{i,m}$. The send and receive events of message M are denoted as *send*(M) and *receive*(M) respectively. The following definitions depict the relations between checkpoints and messages.

DEFINITION 1. *If message M is sent by process P_i before it takes the checkpoint $C_{i,m}$, then we say that send*(M) $\in C_{i,m}$. *If process P_i* takes checkpoint $C_{i,m}$ before it sends out message M, we say that send(M) $\notin C_{i,m}$.

DEFINITION 2. *If message M is received and processed by P_j before P_j takes checkpoint $C_{j,n}$, then we say that receive* (M) $\in C_{j,n}$. *If P_j takes checkpoint $C_{j,n}$ before it receives the message M, we say that receive*(M) $\notin C_{j,n}$.

Given two processes P_i and P_j, suppose the latest (newest) checkpoints of the two processes are $C_{i,m}$ and $C_{j,n}$ respectively. P_j is directly checkpoint dependent on P_i if and only if there exists a message M such that receive(M) $\notin C_{i,m}$ and send(M) $\notin C_{j,n}$. The transitive closure of the direct checkpoint dependency relation is the transitive checkpoint dependency relation. In the remainder of the paper, we do not distinguish between direct and transitive dependency if the context is clear.

It has been shown in [5] that if P_j is checkpoint-dependent on P_i, then when P_i initiates a checkpoint, P_j must also take its corresponding checkpoint in order to ensure global checkpoint consistency.

Among the various approaches to rollback recovery, coordinated checkpointing has been found to be relatively more practical than other techniques due to its simplicity of recovery [15]. Coordinated checkpointing typically operates in three phases. In the first phase when a process P initiates a checkpoint request, P asks all the relevant processes to take their tentative checkpoints. In the second phase, P collects the responses from the relevant processes. In the third phase, if all the relevant processes respond positively by taking their tentative checkpoints, then P sends a final decision to all the relevant processes to turn their tentative checkpoints into permanent ones. Otherwise, the checkpointing operation is aborted and all tentative checkpoints are discarded. A challenging task in this operation is to accurately identify relevant processes. It is most desirable that a checkpoint initiator can identify all the relevant processes at checkpoint time, so that it can inform all of them simultaneously. In Section 4.2, we will describe a technique to address this issue.

Due to the vulnerability of mobile devices, we assume the dependency information among processes is maintained on the fixed hosts. In particular, we store such information for all processes executing in an MH in the stable storages of the MSS which is currently responsible for the MH. We assume all communications to and from an MH pass through its

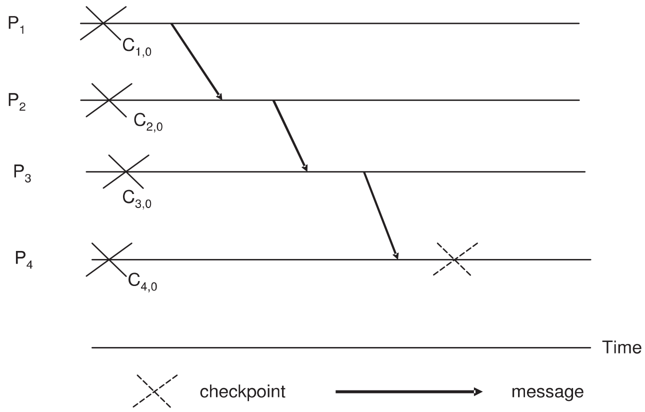

FIGURE 1. An example that illustrates the problem with the sequential checkpointing algorithm of Koo and Toueg [5].

supporting MSS. Whenever a hand-off occurs for an MH, the dependency information for all the processes in the MH is forwarded to the destination MSS.

3.2 Problems with coordinated checkpointing

As we noted above, the coordinated approach must identify which processes are involved in a global checkpoint when a process initiates a checkpoint request. For example, with Koo and Toueg's algorithm [5], when a process P takes a checkpoint, it sends checkpoint requests to all other processes which have sent messages to P since its latest checkpoint. These processes then in turn force other relevant processes to take their checkpoints. As an illustration, consider the four processes shown in Figure 1.

In the beginning, each process P_i, $1 \leq i \leq 4$, takes its initial checkpoints $C_{i,0}$. P_1 sends a message to P_2. Then, P_2 sends a message to P_3. Finally, P_3 sends a message to P_4. Suppose P_4 plans to take a new checkpoint after receiving a message from P_3. Because P_4 receives P_3's message after $C_{4,0}$, it must send a checkpoint request to P_3, which forces P_3 to take a tentative checkpoint. Because P_3 receives P_2's message after $C_{3,0}$, P_3 in turn must ask P_2 to take a tentative checkpoint. By the same token, P_2 must ask P_1 to take a checkpoint after receiving P_3's request. The checkpoint dependency tree in this case is a straight line with four nodes, hence a depth of three.

When P_1 makes a decision regarding its tentative checkpoint, it forwards its decision to P_2. P_2 then sends its own decision to P_3, taking P_1's decision into consideration. Finally, the decision of P_3 is received by P_4. Clearly, this sequential receive-and-forward scheme can result in long checkpointing latency. This problem is particularly serious in a mobile environment in which the bandwidth of the mobile network could be limited. Furthermore, an MH could move from one cell to another; sending a checkpoint request to a process in an MH will need to locate the MH first. This host searching requirement can further increase the time latency for completing a global checkpoint operation.

It is thus our aim to design a coordinated scheme such that each process involved in a global checkpoint can be informed directly by the checkpoint initiator, and every involved process can send its own decision directly to the checkpoint initiator. Hence, the latency time involved in a global checkpointing operation can be greatly reduced. In addition, we want our algorithm to enforce a minimum number of processes to take checkpoints. This property is particulary important for mobile applications again because wireless network has relatively low bandwidth and MHs have relatively low computation power.

4. CONCURRENT CHECKPOINTING

4.1 Main idea

Our primary design goal is to minimize the latency of global checkpointing operation. When a process P initiates a checkpoint, all other processes which are checkpoint dependent upon P must also take their checkpoints in order to maintain global checkpoint consistency. Our algorithm tries to make the checkpoint dependency information available right at the time when a process initiates a checkpoint request.

Our idea is to use a piggyback technique: during normal message transmission, checkpoint dependency information of the sending process is attached with the computation message and sent to the destination process. We associate each process P_j with a set variable RMF_j. When P_j begins execution, RMF_j is initialized to be an empty set. A process P_i is included in RMF_j if there is a sequence of one or more message transmissions from P_i to P_j. In other words, when P_i sends a message to P_j, RMF_j is updated to be $RMF_j \cup RMF_i \cup \{P_i\}$.

Suppose there are $n \geq 2$ processes $P_1, P_2, \ldots, P_n$, and assuming $\forall\, k$, $1 \leq k \leq n-1$, there is a message m_k sent from P_k to P_{k+1}. Also assuming $RMF_n = \{P_1, P_2, \ldots, P_{n-1}\}$. Based on the definition of checkpoint dependency, we have the following observation:

Observation 1. *P_i, $1 \leq i \leq n-1$, is checkpoint-dependent on P_n unless one of the following conditions holds: (i) some process P_k, $i \leq k \leq n-1$, takes a checkpoint after it sends out m_k; (ii) some process P_{k+1}, $i \leq k \leq n-1$, takes a checkpoint after it receives m_k.*

Note that if the second condition in Observation 1 holds for some process P_{i+1}, $1 \leq i \leq n-1$, then the first condition must also hold for process P_i, otherwise m_i becomes an orphan message, and the global state becomes inconsistent. This implies that each element in RMF_n is a potential candidate to take a checkpoint when P_n takes its checkpoint. In Section 4.2, we will describe a technique to determine if an element of RMF_n is truly checkpoint dependent on P_n. Hereafter, if $P_i \in RMF_j$, we say P_i is potentially checkpoint dependent on P_j or P_i is a potential checkpoint dependent of P_j.

It is important to note that RMF_j does reflect accurate information on P_j's potential checkpoint dependents if there

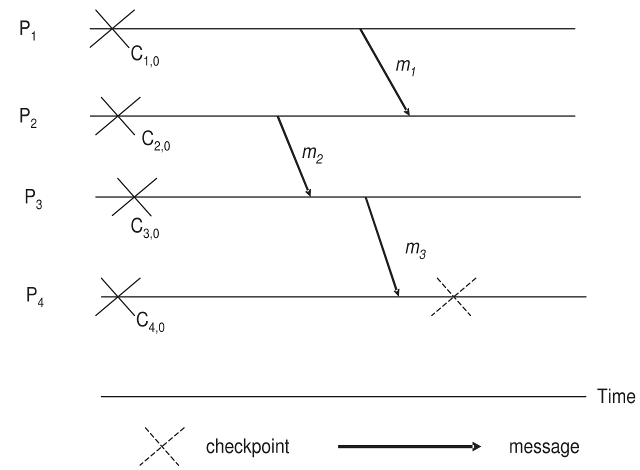

FIGURE 2. An example that illustrates the problem with tardy messages.

is no tardy message in the system. A message is *tardy* if it is received by a process P after P has sent out at least one message. The example shown in Figure 2 explains the problem with using *RMF* to identify potential checkpoint dependents when there are tardy messages in the system.

P_2 first sends a message m_2 to P_3, then P_3 sends a message m_3 to P_4, finally P_1 sends a message m_1 to P_2. Clearly, P_1, P_2 and P_3 are all checkpoint dependent upon P_4. Hence, when P_4 initiates a checkpoint, all of the three processes should take their checkpoints in order to maintain global checkpoint consistency. However, when P_4 initiates a checkpoint, RMF_4 is equal to $\{P_2, P_3\}$, rather than $\{P_1, P_2, P_3\}$. This is because the fact that P_1 has sent a message to P_2 is only recorded in RMF_2, rather than RMF_4, by the time P_4 initiates a checkpoint. In other words, the message m_1 in Figure 2 is a tardy message. This example illustrates that the piggyback technique described above must be enhanced because RMF_i may not include all the processes that are potentially checkpoint dependent on P_i. In the following section, we will describe a technique which allows us to accurately identify all the processes potentially involved in a global checkpoint, even in the presence of tardy messages.

4.2 Dealing with tardy messages

We associate with each process P a message sequence number, $msn(P)$. $msn(P)$ is initialized to zero when P starts, and is incremented by one each time P sends out or receives a message. Hence, $msn(P)$ denotes the number of messages received by P or sent out by P. As we explained in Section 4.1, each $P_i \in RMF_j$ is a potential dependent of P_j. To determine if P_i is a true dependent of P_j, we associate P_i with a variable $msn\_ckpt(P_i)$. $msn\_ckpt(P_i)$ is updated to its current $msn(P_i)$ whenever P_i takes a checkpoint. Because we assume each process P_i takes a checkpoint when it starts, $msn\_ckpt(P_i)$ is initialized to zero.

<u>When P_i is about to send out a meassage to P_j</u>
  $++msn(P_i)$;
  P_i sends out its message together with $msn(P_i)$ and RMF_i;
<u>When P_j receives a message from P_i</u>
  $++msn(P_j)$;
  if $\exists \langle i, m, n\rangle \in RMF_j$ then
    replace $\langle i, m, n\rangle$ with $\langle i, msn(P_i), msn(P_j)\rangle$;
  else $RMF_j = RMF_j \cup \langle i, msn(P_i), msn(P_j)\rangle$;
  for each $\langle k, m, n\rangle \in RMF_i$
    if $\exists \langle k, m', n'\rangle \in RMF_j$ then
      replace $\langle k, m', n'\rangle$ with $\langle k, max(m, m'), msn(P_j)\rangle$;
  else $RMF_j = RMF_j \cup \{\langle k, m, msn(P_j)\rangle\}$;
  end for;

FIGURE 3. Algorithm executed when P_i sends a message to P_j.

We suppose the same assumptions for processes $P_1, P_2, \ldots, P_n$ as given in Section 4.1. Let $msn(P_k, send(m_k))$ and $msn(P_k, before\_send(m_k))$ denote the message sequence numbers of P_k right after and just before P_k sent out m_k to P_{k+1}, $1 \leq k \leq n-1$, respectively. Assuming $msn(P_k, send(m_k)) = msn(P_k, before\_send(m_k)) + 1$. Let $msn(P_{k+1}, receive(m_k))$ and $msn(P_{k+1}, before\_receive(m_k))$ denote the message sequence numbers of P_{k+1} right after and just before P_{k+1} received m_k from P_{k+1}, $1 \leq k \leq n-1$, respectively. We assume $msn(P_{k+1}, receive(m_k)) = msn(P_{k+1}, before\_receive(m_k)) + 1$. We can be assured that process P_k does not take a checkpoint after it sends out m_k provided $msn\_ckpt(P_k)$ is smaller than $msn(P_k, send(m_k))$. Similarly, if $msn\_ckpt(P_{k+1})$ is smaller than $msn(P_{k+1}, receive(m_k))$, then we know P_{k+1} does not take a checkpoint after it received m_k. Based on Observation 1, we have the following observation:

OBSERVATION 2. *P_i, $1 \leq i \leq n-1$, is a true dependent of P_n if $\forall$ k, $i \leq k \leq n-1$, $msn\_ckpt(P_k) < msn(P_k, send(m_k))$ and $msn\_ckpt(P_{k+1}) < msn(P_{k+1}, receive(m_k))$.*

During normal message transmission, each transmitted message is still attached with an *RMF*. However, each element of *RMF* is changed to a different format: if a 3-tuple $\langle k, m, n\rangle$ is included in RMF_j then P_k is potentially checkpoint dependent on P_j. To facilitate the description of our algorithm, if $\langle k, m, n\rangle$ is in RMF_j, we also say P_k is included in RMF_j. Figure 3 shows the algorithm that is executed when P_i sends a message to P_j. When P_j receives a message from P_i, P_i becomes a potential dependent of P_j. As shown in Figure 3, we check if such dependency has occurred before, i.e. some 3-tuple $\langle i, m, n\rangle$ is already in RMF_j. If so, we replace it with the newest dependency information, i.e. with current message sequence numbers of P_i and P_j. Otherwise, we simply record the newest dependency information in RMF_j.

As one can see in Figure 3, we also check if there is any other process P_k who will become potentially dependent on P_j because some 3-tuple $\langle k, m, n\rangle$ is in RFM_i and P_i sends a message to P_j. For each such process, we check if such

Each process P_k:

$CheckpointInProgress_k = \text{FALSE}$;

$NotifyOthers_k = \text{FALSE}$;

$$WillingToCheckpoint_k = \begin{cases} \text{TRUE if } P_k \text{ is willing to take a checkpoint} \\ \text{FALSE otherwise} \end{cases}$$

FIGURE 4. Initialization of system variables.

Checkpoint initiator P_i:

```
Begin
  Willing_Processes = Not_Willing_Processes =
    Non_ Dependent_Processes = ∅;
  for each <j, m, n> ∈ RMF_i
    P_i sends a checkpoint request <i,1/| RMF_i |, m> to P_j
  end for;
  accu_weight = 0;
  while accu_weight < 1 do
    upon receipt of (w, j, Response) DO
      accu_weight = accu_weight + w;
      record P_j in Non_Dependent_Processes, Willing_
        Processes, or Not_Willing_Processes,
      dependent on Response being IGNORE,
        TRUE, or FALSE:
    OD:
  end while:
  if Not_Willing_Processes = = ∅ then
    send ''Take a checkpoint'' to each P_j ∈ Willing_
    Processes;
    P_i itself takes a checkpoint;
  else
    send ''Abort the checkpointing'' to each P_j ∈
    Willing_Processes ∪ Not_Willing_Processes;
  end if:
  send ''No need to take a checkpoint'' to each P_j ∈
  Non_Dependent_Processes –
    (Willing_Processes ∪ Not_Willing_Processes);
End
```

FIGURE 5. Checkpoint algorithm for initiator P_i.

dependency has occurred before, i.e. some 3-tuple $\langle k, m', n' \rangle$ exists in RMF_j. If so, we replace it with the newest dependency information. As shown in Figure 3, we store the current message sequence number of P_j in RMF_j (as its third attribute). But note that we record the maximum of old and new message sequence numbers of P_k in RMF_j (as its second attribute). This is because the new message sequence number of P_k, i.e. m, is not necessarily larger than the old message sequence number of P_k, i.e. m'. If we do not record the larger of the two, then a non-tardy message could be wrongly interpreted as a tardy one by our checkpointing algorithm shown in Figure 6. As one can see in Figure 3, if this is the first time that the dependency of P_k on P_j occurs, we simply store the newest dependency information in RMF_j, i.e. with the message sequence numbers of P_k and P_j.

Our checkpointing algorithm requires a number of system variables, which we show in Figure 4. Whenever a process P_i initiates a checkpoint request, the algorithm shown in Figure 5 is executed. P_i needs to inform its potential dependents regarding this request. Each process informed by P_i will reply with one of the following three responses: willing to take a checkpoint, not willing to take a checkpoint, or not a dependent of P_i. For each replied process, we record it in one of the following initially empty variables: *Willing_Processes*, *Not_Willing_Processes*[1] and *Non_Dependent_Processes*.

As we discussed in Section 4.1, if $\langle j, m, n \rangle$ is in RMF_i, then P_j is known to be potentially checkpoint-dependent on P_i. As shown in Figure 5, P_i sends a checkpoint request to each such process. When P_i sends a request to a potentially dependent process, it also attaches the request message with a weight of $1/|RMF_i|$, where $|RMF_i|$ is the number of elements in RMF_i. If each potential dependent responds to P_i with its received weight and the total weights received by P_i accumulate to one, then P_i knows that all relevant processes have responded. At that time, if all dependent processes are willing to take checkpoints ($Not\_Willing\_Processes = \emptyset$), then each such process is given the instruction to take a checkpoint. If one of the dependent processes replied negatively, then the checkpointing procedure is aborted.

A distinguishing characteristic of our checkpoint approach is that checkpoint requests are concurrently propagated to all potentially dependent processes from the global initiator. When receiving a checkpoint request, process P_j executes the algorithm shown in Figure 6. It is possible that P_j inherits requests from more than one other process. When this occurs, it can invoke more than one copy of the algorithm, which implies proper synchronization must be utilized.

In order to maintain correct checkpoint state, P_j is first blocked from receiving computation messages until current global checkpointing procedure completes. This is done by setting the variable $CheckpointInProgress_j$ to TRUE. As we explained in Observation 1, we must check if the process receiving a checkpoint request is a true dependent of the global checkpoint initiator. This is verified via the check '$msn\_ckpt(P_j) < m$' in our algorithm. Observation 1 includes other conditions, besides '$msn\_ckpt(P_j) < m$', which we will show in Section 5.1 (Lemma 1) can be safely ignored. If P_j is found to be not a dependent of P_i, then P_j lets P_i know that it can be skipped in the checkpointing process.

If P_j is indeed a dependent of P_i and P_j is willing to take a checkpoint, we need to see if there is any process not listed in RMF_i that is dependent on P_i. This situation will arise when there is a dependent process P_j who receives a checkpoint

[1] After all relevant processes reply to P_i, the union of *Willing_Processes* and *Not_Willing_Processes* equals the set of processes that are actually dependent on P_i.

```
Whenever a process P_j receives a checkpoint request
⟨i, w, m⟩;
  Begin
  if not CheckpointInProgress_j then
     CheckpointInProgress_j = TRUE; //Signal blocking of P_j
  end if;
  if msn_ckpt(P_j) < m then // P_j is a true dependent of P_i
     if WillingToCheckpoint_j then
        if not TestAndSet(NotifyOthers_j) then //Ensure P_j
        notifies its dependents at most once
           for each 3-tuple <k, m', n'> ∈ RMF_j and n' > m do
              P_j forwards a checkpoint request to P_k
              with a weight WT_j;
           end for;
        end if;
        P_j responds to P_i with (WT_j, j, WillingToCheckpoint_j);
  else   //P_j is not a dependent of P_i
        P_j responds to P_i with (w, j, IGNORE);
  end if;
End

When P_j receives the final decision:
  Act according to received instruction;
  if P_j takes a checkpoint then
     msn_ckpt(P_j) = msn(P_j)
  end if;
  NotifyOthers_j = FALSE;
  CheckpointInProgress_j = FALSE;
```

FIGURE 6. Checkpoint algorithm for all potentially dependent processes.

request $\langle i, w, m\rangle$, and its RMF_j contains a 3-tuple $\langle k, m', n'\rangle$ such that $n' > m$. $n' > m$ implies that some message m_1 must be received by P_j after another message m_2 was sent out by P_j, and m_2 eventually leads to the 3-tuple $\langle i, w, m\rangle$ being recorded in P_i or some other process. Hence, m_1 is a tardy message. In this case, P_j will serve as an agent for P_i by notifying other potentially dependent processes. We divide the weight received by P_j, i.e. w, among P_j and all other processes notified by P_j. This divided weight, denoted as WT_j in Figure 6, is sent together with the request to each potential dependent.

Note that when P_j receives two or more requests, the algorithm in Figure 6 will be invoked more than once. In this case, those potential dependents of P_j are notified more than once. We avoid this by using a TestAndSet instruction [16] which checks and modifies the variable $NotifyOthers_j$ atomically. This scheme also tackles the problem of cyclic checkpoint dependency, which occurs when a process is transitively dependent on itself. In this case, a checkpoint request can be forwarded from one process back to itself. If P_j is involved in a cyclic dependency, it sets $NotifyOthers_j$ to TRUE the first time it notifies its potential dependents. When the request is propagated back to P_j, it can avoid another round of the notification process by checking the value of $NotifyOthers_j$.

As shown in Figure 6, each notified process responds to the checkpoint initiator P_i directly, instead of forwarding their responses through a chain of intermediary processes as in [5]. This is another characteristic of our approach, which potentially reduces the latency of the entire checkpointing process. When the weights received by the global initiator accumulate to one, it knows all relevant processes have replied. As shown in Figure 5, if all dependent processes are willing to take checkpoints (*Not_Willing_Processes* is $\emptyset$), then the global initiator tells them to act accordingly; otherwise, the checkpointing procedure is aborted. Finally, all non-dependent processes are also informed. Note that a process could receive multiple checkpoint requests along different message transmission paths and reply to the global initiator with different responses (mainly because a checkpoint has been taken with respect to a particular path between the process and the global initiator). As long as one of the responses is either willing or not willing to take a checkpoint, then the process is dependent on the global initiator, hence excluded from *Non_Dependent_Processes* as shown in Figure 5.

5. DISCUSSIONS

In this section, we first discuss a few correctness properties of our concurrent checkpoint algorithms in Section 5.1. We then consider the expected performance of our algorithm in terms of the expected length of checkpoint request path in Section 5.2.

5.1 Correctness proofs

In order to better understand the characteristics of our algorithms, we describe a way to group checkpoint dependent processes. Given n processes $P_1, P_2, \ldots, P_n$, we assume there is a message transmission path[2]: a message m_k is sent from P_k to P_{k+1}, $\forall k$, $1 \leq k \leq n-1$. Hence, P_k is potentially checkpoint dependent on P_{k+1}, $1 \leq k \leq n-1$. We partition these n processes into one or more groups according to how each message is sent from one process to another. We place P_1 and P_2 in the first group. We then place P_3 in the same group of P_1 and P_2 if m_2 is sent after P_2's receiving of m_1; otherwise (i.e. m_1 is a tardy message), we place P_3 in a new group. Similarly, we place P_4 in the same group of P_3 if m_3 is sent after P_3's receiving of m_2; otherwise (i.e. m_2 is a tardy message), we place P_4 in a new group. We continue this grouping procedure for all the remaining processes.

Suppose we end up with l groups of processes. The two extreme cases are when $l = 1$ or $l = n - 1$. The first case corresponds to the situation when none of the $n - 1$ messages

[2]In reality, there can be multiple messages transmitted from P_k to P_{k+1}. With our piggyback algorithm shown in Figure 3, it is the last message sent from P_k to P_{k+1} which determines the 3-tuple to be included in RMF_{k+1}.

is tardy, and the second case indicates that every of the $n-1$ messages except the last message m_{n-1} is tardy. In general, we have l groups of processes denoted as $(P_1, \ldots, P_{G_1})$, $(P_{G_1+1}, \ldots, P_{G_2}), \ldots, (P_{G_{l-1}}+1, \ldots, P_{G_l} = P_n)$, where each G_j, $1 \le j \le l$, is an integer that lies between 2 and n. As an example, the four processes shown in Figure 2 are partitioned into two groups: (P_1, P_2) and (P_3, P_4). With our algorithm shown in Figure 3, we have the following observation:

OBSERVATION 3. *RMF_{G_j}, $2 \le j \le l$, includes $P_{G_{j-1}}$ and all processes in group j except P_{G_j}. RMF_{G_1} includes all processes in group 1 except P_{G_1}.*

Furthermore, because the message sent from $P_{G_{j-1}-1}$ to $P_{G_{j-1}}$, $2 \le j \le l$, is tardy, we obtain the following observation:

OBSERVATION 4. *If $\langle G_{j-1}, m, n\rangle$ is a 3-tuple in RMF_{G_j}, then for every 3-tuple $\langle k, m', n'\rangle$ in $RMF_{G_{j-1}-1}$ it must be the case that $n' > m$.*

Based on our checkpoint algorithms shown in Figures 5 and 6, if P_n is the process which initiates a checkpoint request, then the following observation can be made:

OBSERVATION 5. *P_n simultaneously informs all processes in RMF_n (i.e. RMF_{G_l}) to take their tentative checkpoints. P_{Gj}, $1 \le j \le l-1$, is the process which on P_n's behalf simultaneously informs all processes in RMF_{G_j} to take their tentative checkpoints. P_{G_j}, $1 \le j \le l-1$, does not start its notification of processes in RMF_{G_j} until it is informed by $P_{G_{j+1}}$.*

We call P_{G_j} the *group* checkpoint initiator for group j, and P_n the *global* checkpoint initiator. Consider the example in Figure 2 again, if P_4 issues a checkpoint request, then P_2 and P_4 will act as group initiators for the first and second groups of processes respectively. At checkpoint time, RMF_4 contains a 3-tuple $\langle 2, 1, 1\rangle$ whose second attribute is smaller than the third attribute of the 3-tuple $\langle 1, 1, 2\rangle$ which is included in RMF_2. Hence, P_2, after receiving a request from P_4, will notify P_1 on behalf of P_4.

Note that if there exist other messages besides the $n-1$ messages described above, then a process may be notified by more than one other process. Specifically, if a process appears in more than one group initiator's *RMF*, then that process will inherit requests from more than one group initiator. This situation can occur if, on a message transmission path, an additional message is sent from one process in a group to another process in a different group. More generally, there could exist more than one message transmission path between a process and the global initiator. Hence, a process could inherit requests forwarded along different paths from the global initiator. Whenever a request is received by a process, it will reply to the global initiator with a proper response, together with its received weight. Each process, upon receiving the final decision from the global initiator (made after the total weights received by the global initiator sum to one), will take at most one checkpoint during a global checkpointing operation.

When P_n initiates a global checkpoint request, the algorithm shown in Figure 6 executes the check '$msn\_ckpt(P_j) < m$' to determine if P_j is a true dependent of P_n. Note that m is equal to $msn(P_j, send(m_j))$, i.e. the message sequence number of P_j when P_j sent out m_j. According to Observation 2, we should also verify '$msn\_ckpt(P_k) < msn(P_k, send(m_k))$' and '$msn\_ckpt(P_{k+1}) < msn(P_{k+1}, receive(m_k))$', $\forall_k, j+1 \le k \le n-1$. The following lemma tells us that if $msn\_ckpt(P_j) < msn(P_j, send(m_j))$, then we can be assured of the following two facts: (i) $msn\_ckpt(P_k) < msn(P_k, send(m_k))$, $\forall k, j+1 \le k \le n-1$; and (ii) $msn\_ckpt(P_{k+1}) < msn(P_{k+1}, receive(m_k))$, $\forall\, k, j \le k \le n-1$.

LEMMA 1. *When P_j is requested to respond because P_n initiates a global checkpoint request, it is sufficient to check $msn\_ckpt(P_j) < msn(P_j, send(m_j))$ in order to verify that P_j is a true dependent of P_n.*

Proof. We prove by contradiction. Suppose $msn\_ckpt(P_j) < msn(P_j, send(m_j))$, but $msn\_ckpt(P_k) \ge msn(P_k, send(m_k))$ for some process P_k, $j+1 \le k \le n-1$. This means that P_k has taken a checkpoint after sending out m_k. If none of the messages $m_j, m_{j+1}, \ldots, m_{k-1}$ is tardy, then all of the processes $P_{k-1}, P_{k-2}, \ldots, P_j$ must have been requested to take their checkpoints when P_k took its checkpoint after sending out m_k. This means that $msn\_ckpt(P_j) \ge msn(P_j, send(m_j))$, which contradicts our assumption. Suppose a message m_l, $k-1 \le l \le j$, is tardy. P_{l+1} should have been requested to take a checkpoint when P_k took its checkpoint after sending out m_k. This means that P_{l+1} will not send out checkpoint request when P_n initiates its checkpoint request. This implies that P_j should not have received a checkpoint request, which again contradicts our premise.

Using similar reasoning, we can show that if $msn\_ckpt(P_j) < msn(P_j, send(m_j))$, then $msn\_ckpt(P_{k+1}) < msn(P_{k+1}, receive(m_k))$, $\forall k, j \le k \le n-1$. □

Note that if P_j receives checkpoint requests from more than one initiator, then the condition '$msn\_ckpt(P_j) < msn(P_j, send(m_j))$' must hold for at least one message transmission path (from P_j to P_n) in order for P_j to be dependent on P_n.

LEMMA 2. *Every process terminates its execution of the checkpoint algorithms shown in Figures 5 and 6.*

Proof. Two situations can prevent a process from terminating: (i) it keeps on receiving and forwarding checkpoint requests; (ii) it waits for the global initiator's decision *ad infinitum*. The first case may occur when a process is involved in a cyclic checkpoint dependency. However, our algorithm shown in Figure 6 avoids this because it allows a process to forward requests to its potential dependents at most once. When a process initiates a checkpoint request, each process that is potentially dependent on the global initiator will receive a portion of a weight that is initialized to one. Each such process will directly reply to the global initiator with its received weight. When the weights received by the global initiator sum to one, an appropriate instruction will be sent to each replied process. All these messages will be delivered to the corresponding processes, because we assume messages are exchanged through reliable channels. Thus processes do not wait forever for replies from the global initiator. □

Let $DependentProcesses_n$ denote the set of processes that are checkpoint dependent on P_n. Suppose all processes in $DependentProcesses_n$ are willing to take checkpoints when P_n initiates a checkpoint request. The following theorem describes an important characteristic of our checkpoint algorithms shown in Figures 5 and 6:

Theorem 5.1. *Process P_j will take a checkpoint when P_n initiates a checkpoint request if and only if $P_j \in DependentProcesses_n$.*

Proof. (The if part) Based on Observations 3, 4 and 5, we know P_j will be requested by some group initiator P_{Gk} to take a checkpoint when P_n initiates a checkpoint request. Based on Observation 2, we have $\forall h, j \le h \le n-1$, $msn\_ckpt(P_h) < msn(P_h, send(m_h))$ and $msn\_ckpt(P_{h+1}) < msn(P_{h+1}, receive(m_h))$. Since $msn\_ckpt(P_j) < msn(P_j, send(m_j))$, P_j will reply to P_n with a response 'willing to take a checkpoint', and will take a checkpoint when it receives the final decision from P_n.

(The only if part) If P_j takes a checkpoint after P_n initiates a checkpoint request, then the following two conditions must hold: (i) P_j must appear in RMF_{G_k} for some group initiator P_{G_k}; (ii) $msn\_ckpt(P_j) < msn(P_j, send(m_j))$. Based on Lemma 1, we know $msn\_ckpt(P_h) < msn(P_h, send(m_h))$, $\forall h, j+1 \le h \le n-1$, and $msn\_ckpt(P_{h+1}) < msn(P_{h+1}, receive(m_h))$, $\forall h, j \le h \le n-1$. Based on Observation 3, there exists a sequence of group initiators, $P_{G_k}, P_{G_{k+1}}, \ldots, P_{G_l}$, such that P_{G_h} appears in $RMF_{G_{h+1}}$, $\forall h, k \le h \le l-1$, and P_{G_l} is P_n. According to Observation 2, we know processes in RMF_{G_h} are dependent on P_{G_h}, $\forall h, k \le h \le l$. Because P_{G_h} is in $RMF_{G_{h+1}}$, $\forall h, k \le h \le l-1$, P_j is dependent on P_n as was to be proved. □

Owing to Theorem 5.1 and the fact that each process notified by the global initiator takes at most one checkpoint, the following two corollaries follow.

Corollary 1. *The number of processes that take new checkpoints during the execution of our checkpoint algorithms is minimal.*

Corollary 2. *If the set of checkpoints in the system is consistent before the execution of our checkpoint algorithms, then set of checkpoints in the system is consistent after the algorithms terminate.*

5.2 Expected performance analysis

Based on our discussions in Section 5.1, we know it is the number of group initiators on a message transmission path which determines the latency in propagating a checkpoint request from the global initiator to a potentially dependent process. When there are multiple paths between a process and the global initiator, the path with the largest number of group initiators requires the longest time for request propagation if all other conditions for the paths are identical. In this section, we study the expected performance of our concurrent checkpoint algorithm, and compare that with the performance of Koo and Toueg's sequential checkpointing algorithm [5]. Consider again the same message transmission patterns among the n processes $P_1, P_2, \ldots, P_n$ as described in Section 5.1. Suppose P_k is dependent on P_{k+1}, $1 \le k \le n-1$. Recall that with Koo and Toueg's algorithm, if P_n initiates a checkpoint request, then the request will be propagated from P_n to P_1, passing through all intermediary processes P_{n-1}, $P_{n-2}, \ldots, P_3$, P_2. In this case, the relationships among the processes form a checkpoint request path, with the n processes being the vertices on the path. Hence, the length of the path is $n-1$, which we denote as $L_{kt}(n)$.

Now we consider the length of the checkpoint request path for our algorithm, which we denote as $L_c(n)$. Consider the message m_{n-1} that is sent from P_{n-1} to P_n, and m_{n-2} that is sent from P_{n-2} to P_{n-1}. If P_{n-1} sends m_{n-1} to P_n after P_{n-1} receives m_{n-2} from P_{n-2}, then $L_c(n) = L_c(n-1)$. This is because in this case P_n, P_{n-1} and P_{n-2} will be placed in the same group, and will be notified simultaneously by the same group initiator. Hence, the checkpoint request path when P_n initiates a checkpoint request and the checkpoint request path when P_{n-1} initiates a checkpoint request must have the same length. However, If P_{n-1} sends m_{n-1} to P_n before P_{n-1} receives m_{n-2} from P_{n-2}, then $L_c(n) = L_c(n-1) + 1$. Assuming p is the probability that P_{k-1} sends m_{k-1} to P_k after P_{k-1} receives m_{k-2} from P_{k-2}, and $1-p$ is the probability that P_{k-1} sends m_{k-1} to P_k before P_{k-1} receives m_{k-2} from P_{k-2}, $\forall k$, $3 \le k \le n$. We derive the expected value of $L_c(n)$, denoted as $E[L_c(n)]$, as follows:

$$
\begin{aligned}
&E[L_c(n)]\\
&= p \times L_c(n-1) + (1-p) \times (L_c(n-1) + 1)\\
&= L_c(n-1) + (1-p)\\
&= p \times L_c(n-2) + (1-p) \times (L_c(n-2) + 1)\\
&= L_c(n-2) + (1-p) \times (1-p) = \ldots\\
&= L(2) + (n-2) \times (1-p)\\
&= 1 + (n-2) \times (1-p)
\end{aligned}
$$

Note that when p approaches 1, i.e. most messages are non-tardy, $E[L_c(n)] \approx 1$. This certainly is the condition under which our algorithm performs best. We compare the performance of our algorithm and Koo and Toueg's algorithms by computing the ratio: $E[L_c(n)]/E[L_{kt}(n)] = [1 + (n-2) \times (1-p)/n - 1$. When n is very large, the ratio approaches $1-p$. If p is 0.5, then our algorithm is expected to outperform Koo and Toueg's by reducing the latency associated with checkpoint request propagation by 50%. On the other hand, if p approaches 1, then $E[L_c(n)]/E[L_{kt}(n)] = 1/(n-1)$. In this

TABLE 1. System parameters and default settings.

Parameter	Setting	Meaning
N_{mh}	16	Number of MHs
B_{wl}	100 Kbps	Wireless bandwidth
B_{wired}	10 Mbps	Wired bandwidth
I_c	1000 s	Time between two successive global checkpoints
I_{mt}	500 s	Time between two successive transmissions of computation messages
S_{cm}	2 KB	Size of a computation message
S_{sm}	100 bytes	Size of a checkpoint coordination message
S_{rmf}	10 bytes	Size of an *RMF* 3-tuple
T_{sc}	2.5 ms	Time to save a tentative checkpoint in main memory

case, the longer the checkpoint request path, the larger the difference between the performance of our and Koo and Toueg's algorithms.

6. EXPERIMENTAL EVALUATION

This section presents the performance evaluation of our approach via simulation experiments. We implemented an event-driven simulator so that we could perform experiments. The simulator is written in C and runs on a Windows PC. System settings are controlled by the parameters listed in Table 1.

6.1 Simulation model and performance metrics

Our simulation environment comprises a number of MSSs and MHs. We assume there is one process running on each MH. The MSSs are connected by a wired network with a bandwidth of 10 Mbps. Each MH has a wireless connection with its supporting MSS with a bandwidth of 100 Kbps. The size of each computation message is assumed to be 2 KB. The size of an *RMF* 3-tuple, i.e. the control information piggybacked on a computation message, is 10 bytes. During checkpointing, there are coordination messages transmitted among processes for checkpoint requests and responses. Each coordination message is assumed to be 100 bytes. We assume the time needed to save a tentative checkpoint in main memory to be 2.5 ms.

To model the processes' sending of computation messages, the Poisson process is used. In other words, if τ is the time between two successive message sending events, then τ is an exponentially distributed random variable. Both the sender and the receiver are selected at random. When a computation or coordination message is sent from one process to another, the message is received and then forwarded to the destination host by the supporting MSSs. Thus, the transmission delay for a computation message is $2 \times [8 \times (S_{cm}/B_{wl}) + 8 \times (S_{cm}/B_{wired})$, i.e. $2 \times (8 \times 2/100) + 8 \times 2/10\,000 = 322$ ms, where S_{cm}, B_{wl} and B_{wired} are the size of a computation message, wireless bandwidth and wired bandwidth, respectively. The time needed to send a checkpoint coordination message is $2 \times [8 \times (S_{sm})/(B_{wl}) + 8 \times (S_{sm})/(B_{wired})$, i.e. $2 \times (8 \times 0.1/100) + 8 \times 0.1/10\,000 = 16$ ms, where S_{sm} is the size of a coordination message. We assume the time between two successive global checkpoints to be constant (1000 s). At scheduled checkpointing time, an active process is randomly selected as the global checkpoint initiator.

The primary performance metric in our simulation is blocking time, which is the time duration from a global checkpoint initiation until its completion. Because our approach achieves lower blocking by having processes carry extra control information, we also measure these overheads, which we call the control information overheads. These overheads are calculated as the ratio of the amount of piggybacked information to that of normal computation messages. During checkpointing, coordination messages are transmitted among processes. We measure the coordination message overheads as the average of such messages per global checkpoint.

Our scheme is designed mainly to improve the checkpointing latency experienced in Koo and Toueg's protocol [5]; their approach naturally becomes our comparison target. In addition, we also compare with the checkpointing protocol of Kim and Park [17]. Unlike other coordinated checkpointing protocols, in which the checkpoint initiator collects the status information of its dependent processes and delivers its decision, the process in Kim and Park's protocol takes a checkpoint when it knows that all its dependent processes took their checkpoints. With this approach, the initiator does not always have to deliver its decision after it collects the status of the dependent processes, hence partially eliminating the third phase of the notify–respond–react three phases commonly seen in other coordinated approaches. As a result, its blocking time can be shortened in certain cases. However, because Kim and Park's protocol still sends checkpoint request and response messages in a sequential receive-and-forward fashion as in Koo and Toueg's protocol, its overall performance must be seen via experiments, which we present in Section 6.2.

For each approach tested in our experiments, 20 simulation runs with different random number seeds are conducted and performance statistics are collected and averaged over the 20 runs. Each simulation run lasts for 1 000 000 s, during which 999 global checkpoints are taken. With this number of checkpoints taken, performance results were observed to stabilize.

6.2 Experimental results

In this section, we present our experimental results. For ease of illustration, we use the abbreviation KT for Koo and Toueg's protocol and KP for Kim and Park's protocol. In our first

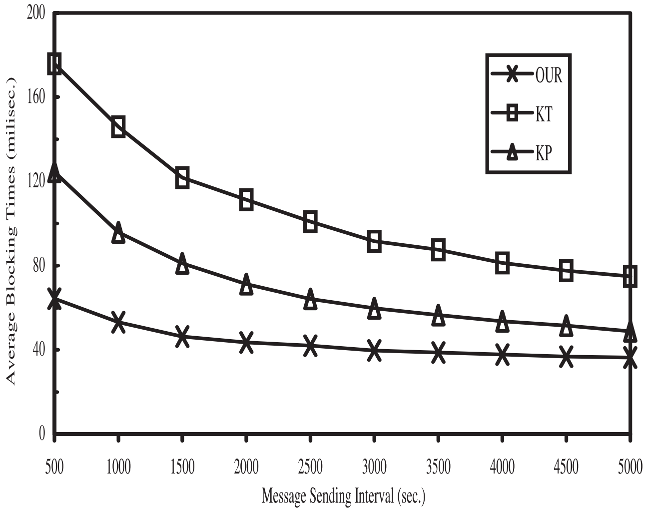

FIGURE 7. Average blocking times versus computation message sending interval.

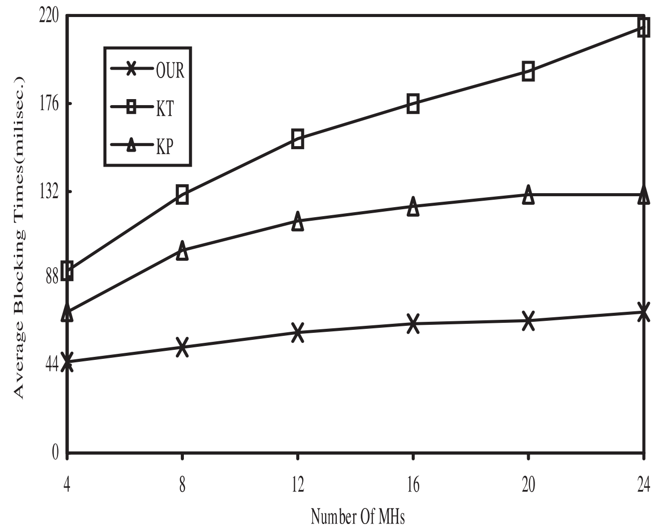

FIGURE 8. Average blocking times versus number of MHs.

experiment, we focus on how different approaches fare in the blocking time metric. We first vary the interval between two successive transmissions of computation messages from 500 to 5000 s in 500 s increments. The other parameters have the base values given in Table 1. The result is illustrated in Figure 7. Observe that for an interval of 500 s, the average blocking times for KT, KP and our algorithm are 175.8, 124.5, and 64.3 ms respectively. Much to our expectation, KT performs worst because it informs the dependents of a global checkpoint initiator in a sequential receive-and-forward fashion. The responses from the dependents follow the same pattern, albeit in the reverse direction. Hence, the blocking time is determined by the longest path from the initiator to its descendants in the checkpointing tree.

KP performs better than KT. This is due to its ability to commit processes located in a subtree locally. However, like KT, KP also uses the sequential notification policy, hence its improvement in blocking time is limited. In contrast to KT and KP, our algorithm simultaneously informs an initiator's dependents, and the dependents directly reply to the initiator, hence the blocking time is reduced significantly. Note in Figure 7 that as the interval becomes larger, the difference among the three approaches becomes less obvious. This can be observed as follows. As the interval increases, the number of messages transmitted among the processes decreases. In this case, the depth of the checkpointing tree is likely to reduce. As a result, the blocking time reduces for all three approaches. Note that the slope of reduction in blocking time for our approach is not as large as that for the other two approaches. This is certainly because our approach is not so sensitive to the depths of the checkpointing trees.

Figure 8 shows that our approach outperforms KT and KP in blocking times with varying number of MHs. As one can see, when the number of MHs increases, the blocking times increase for all three algorithms. Larger number of MHs implies that longer paths could form in the checkpointing tree, which in turn leads to larger blocking times. KT is again more sensitive than KP in this aspect. Our algorithm scales well as the number of MHs increases.

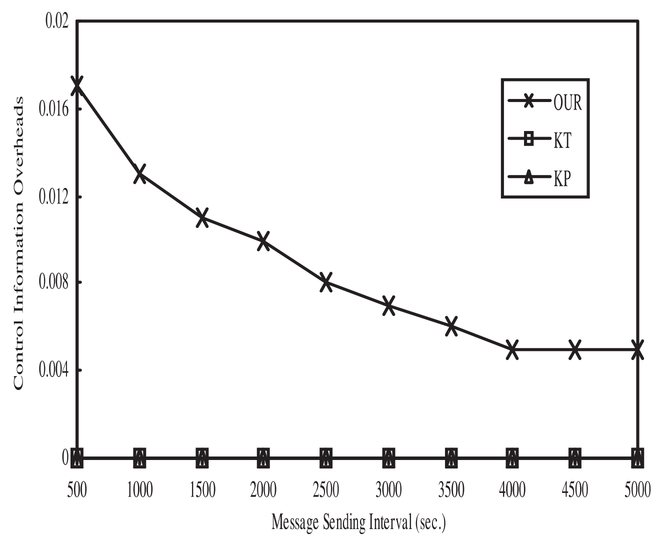

FIGURE 9. Control information overheads versus message sending interval.

Our algorithm does not achieve reduced blocking for free. During normal message transmission, processes must carry extra control information in the form of 3-tuples. In the following two experiments, we measure control information overheads as the ratio of the amount of piggybacked information to that of normal computation messages. Figures 9 and 10 illustrate these costs from two different angles. KT carries no extra control information. With KP, a couple n-bit arrays (n is the number of processes in the system) must be kept

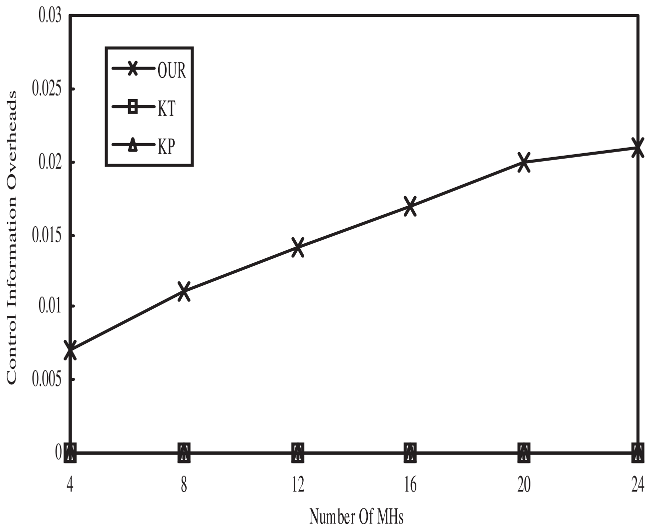

FIGURE 10. Control information overheads versus number of MHs.

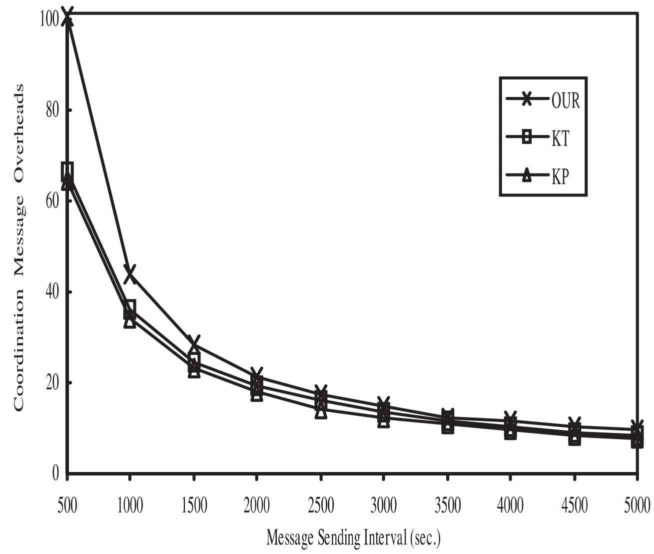

FIGURE 11. Coordination message overheads versus message sending interval.

for housekeeping information, and must be sent along with normal computation messages. Because this cost is rather small, we assume it is negligible in our simulation.

In Figure 9, the overheads in our algorithm decrease as the interval between two successive transmissions of computation messages increases. This is certainly because less computation messages are sent when the interval increases, which implies less *RMF* 3-tuples will be transmitted. Figure 10 shows that the overheads increase as the number of MH increases. Both figures indicate that our algorithm incurs <2% overheads in carrying extra control information for different parameter settings we consider.

One common characteristic of coordinated protocols is that redundant checkpoint request messages are possible. This situation occurs when there are multiple transmission paths from a dependent process to the global initiator. In this case, the dependent process can receive more than one request from other processes, albeit at most one checkpoint will be taken by the dependent process. Compared with KT and KP, our algorithm can induce more of such messages, particularly when a special type of tardy message exists in the system.

Consider Figure 2 once again. m_1 is a tardy message in that example. If P_2 receives an additional message m_4 from P_1 before P_2 sends out m_2, then both P_2 and P_4, rather than just P_2 or P_4, will inform P_1 when P_4 initiates a checkpoint by using our algorithm. By contrast, only P_2 will inform P_1 by using either KT or KP. We want to point out that our algorithm can be revised to avoid this problem by having each group checkpoint initiator to transmit the information concerning the set of processes the group initiator is going to inform on each checkpoint request message. When a process receives a checkpoint request from a group initiator, it can ignore notifying those processes appearing in the set of processes received from the initiator. It is not hard to see that this then becomes a trade-off between reducing the number of request messages and reducing the amount of information carried on each request message.

To understand the behavior of the three algorithms in this respect, we conduct two experiments below. The performance index we use is the average number of coordination messages (including checkpoint requests, responses and final decision) transmitted per global checkpoint, which we call the coordination message overheads.

Figure 11 shows the coordination message overheads for all three algorithms with varying message sending interval. KP has a little performance gain over KT because some processes can locally commit as a group, meaning that they need not wait for the decision from their parents in the checkpointing tree. Compared with KT and KP, our algorithm sends about 40 more coordination messages per global checkpoint when the message sending interval is 500 s. Our overheads become comparable with those of KT and KP when the interval is >1000 s.

Figure 12 shows the coordination message overheads for all three algorithms with different number of MHs. As the figure illustrates, when the number of MHs increases, the overheads increase for all three algorithms. Our algorithm is more sensitive in this case. Note that these extra checkpoint requests do not appear to affect much our algorithm's performance in checkpoint latency, as Figure 8 illustrates. This is because checkpoint latency mainly depends on how coordination messages are transmitted on a checkpoint tree. As we explained in Section 5.1, it is tardy messages that determine overall checkpoint latency in our algorithm. Consider again the example we discussed above. If there were no

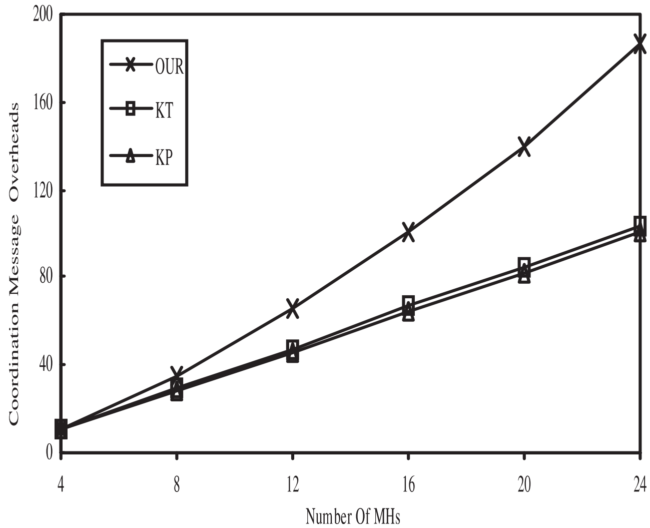

FIGURE 12. Coordination message overheads versus number of MHs.

message m_4 sent from P_1 to P_2 (i.e. we are back to the situation shown in Figure 2), then there will not be any redundant requests when P_4 initiates a checkpoint request. However, the length of the longest checkpoint request path in this case is still 2. While sending redundant requests requires extra communication bandwidth, it affects our algorithm's performance in checkpoint latency to a limited degree.

7. CONCLUSIONS

This paper presents a coordinated checkpointing scheme which reduces the delay involved in a global checkpointing process for mobile computing systems. Reducing such delay is important in all kinds of distributed systems; it is critical in mobile systems due to the mobility of MHs and the limited bandwidth of wireless network. The idea is to collect and store process dependency information when processes exchange computation messages. Processes then use such information at checkpointing time to propagate checkpoint requests to dependent processes without having to trace the dependency tree. The number of processes that take new checkpoints during the execution of our checkpoint algorithm is shown to be minimal. This property is particulary important for mobile applications because the wireless network has low bandwidth and the MHs have relatively low computation power. Via probability-based analysis, we show that our scheme can reduce the latency associated with checkpoint request propagation by 50%, compared with traditional coordinated checkpointing approaches. Experimental results indicate that we have <2% overhead in transmitting piggybacked information during normal runtime. However, we can achieve up to a 60% reduction in checkpoint latency time.

ACKNOWLEDGEMENTS

This paper is an extended and revised version of a paper that appeared in the Proceedings of the 29th IEEE Annual International Computer Software and Applications Conference, 2005. This work was supported, in part, by NSC grant 93-2213-E-006-097 and NSF grant 60203017. We thank Hongya Wang at Huazhong University of Science and Technology for implementing the simulation programs and performing the experiments. We also thank the anonymous referees for their valuable comments.

REFERENCES

[1] Cao, G. and Singhal, M. (2001) Mutable checkpoints: a new checkpointing approach for mobile computing systems. *IEEE Trans. Parall. Distr. Syst.*, **12**(2), 157–172.

[2] Park, T., Woo, N. and Yeom, H. Y. (2002) An efficient optimistic message logging scheme for recoverable mobile computing systems. *IEEE Trans. Mobile Comput.*, **1**(4), 265–277.

[3] Lin, C. Y., Wang, S. C. and Kuo, S. Y. (2003) An efficient time-based checkpointing protocol for mobile computing systems over mobile IP. *Mobile Netw. Appl.*, **8**, 687–697.

[4] Acharya, A. and Badrinath, B. R. (1994) Checkpointing distributed applications on mobile computers. In *Proc. 3rd Int. Conf. Parallel and Distributed Information Systems*, Austin, TX, September, pp. 73–80, IEEE.

[5] Koo, R. and Toueg, S. (1987) Checkpointing and roll-back recovery for distributed systems. *IEEE Trans. Softw. Eng.*, **SE-13**(1), 23–31.

[6] Elnozahy, E. N., Alvisi, L., Wang, Y. M. and Johnson, D. B. (2002) A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, **34**(3), 375–408.

[7] Pradhan, D. K., Krishna, P. and Vaidya, N. H. (1996) Recoverable mobile environment: design and trade-off analysis. *Proc. 26th Int. Symp. Fault-Tolerant Computing Systems*, Sendai, Japan, June, pp. 16–25, IEEE.

[8] Chen, I.-R., Gu, B., George, S. E. and Cheng, S.-T. (2005) On failure recoverability of client–server Applications in mobile wireless environments. *IEEE Trans. Reliab.*, **54**(1), 115–122.

[9] Cao, G. and Singhal, M. (1998) On the impossibility of min-process non-blocking checkpointing and an efficient checkpointing algorithm for mobile computing systems. *Proc. 27th Int. Conf. Parallel Processing*, Minneapolis, August, pp. 37–44, IEEE.

[10] Kumar, L., Mishra, M. and Joshi, R. C. (2003) Low overhead optimal checkpointing for mobile distributed systems. *Proc. 19th Int. Conf. Data Engineering*, Bangalore, India, March, pp. 686–688, IEEE.

[11] Manabe, Y. (2001) A distributed consistent global checkpoint algorithm for distributed mobile systems. *Proc. 8th Int. Conf. Parallel and Distributed Systems*, KyongJu City, Korea, June, pp. 26–29, IEEE.

[12] Neves, N. and Fuchs, W. K. (1997) Adaptive recovery for mobile environments. *Commun. ACM*, **40**(1), 68–74.

[13] Yao, B., Ssu, K. F. and Fuchs, W. K. (1999) Message logging in mobile computing. *Proc. 29th Int. Symp. Fault-Tolerant Computing Systems*, Madison, Wisconsin, June, pp. 294–301, IEEE.

[14] Randell, B. (1975) System structure for software tolerance. *IEEE Trans. Softw. Eng.*, **1**(2), 220–232.

[15] Elnozahy, E. N. and Plank, J. S. (2004) Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secure Comput.*, **1**(2), 97–108.

[16] Silberschatz, A., Galvin, P. B. and Gagne, G. (2002) *Operating System Concepts*. 6th ed. John Wiley & Sons, Inc., New York.

[17] Kim, J. L. and Park, T. (1993) An efficient protocol for checkpointing recovery in distributed systems. *IEEE Trans. Parall. Distr. Syst.*, **4**(8), 955–960.