A Low Overhead Checkpointing Protocol for Mobile Computing Systems

Chi-Yi Lin, Szu-Chi Wang, Sy-Yen Kuo

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
sykuo@cc.ee.ntu.edu.tw

Ing-Yi Chen

Department of Electronic Engineering
Chung Yuan Christian University
Chung-Li, Taiwan
ichen@cycu.edu.tw

Abstract

Checkpointing protocols for distributed computing systems can also be applied to mobile computing systems, but the unique characteristics of the mobile environment need to be taken into account. In this paper, an improved time-based checkpointing protocol is proposed, which is suitable for mobile computing systems based on Mobile IP. The main improvement over a traditional time-based protocol is that our protocol reduces the number of checkpoints per checkpointing process to nearly minimum, so that fewer checkpoints need to be transmitted through the bandwidth-limited wireless links. The proposed protocol also performs very well in the aspects of minimizing the number and the size of messages transmitted in the wireless network. Therefore, the protocol brings very little overhead to a mobile host which has limited resource. Additionally, by integrating the improved timer synchronization technique, our protocol can also be applied to wide area networks.

1. Introduction

The infrastructure supporting mobile computation is growing mature rapidly. Users with mobile devices are able to access and exchange information on the move. As a result, collaborative works can be done effectively, no matter where the participating members/hosts are physically located. For example, in a sensor network which carries out real-time scientific computation, sensors with processing capability can be mobile and distributed.

To provide fault-tolerance capability for mobile computing systems, checkpointing and rollback-recovery techniques for traditional distributed computing systems such as [1, 2] can be used. Recently, checkpointing protocols specifically designed for mobile computing systems have also been proposed [3-11]. A common goal of these protocols is to avoid extra coordinating messages and unnecessary checkpoints. Prakash and Singhal [4] first proposed a checkpointing protocol that requires only a minimum number of processes to take checkpoints and does not block the underlying computation during checkpointing. However, Cao and Singhal [6] proved that such a min-process nonblocking checkpointing algorithm does not exist. They also introduced the concept of *mutable checkpoints* [10] in their nonblocking algorithm, which forces a minimum number of processes to take checkpoints on the stable storage.

Time-based protocols [2, 5, 12] use synchronized clocks or timers to indirectly coordinate the creation of checkpoints so that coordinating messages are reduced. However, time-based protocols require *every* process to take a checkpoint during a checkpointing process. Moreover, since timers cannot be perfectly synchronized, the consistency between all the checkpoints can still be a problem. In [12], the problem is solved by disallowing message sending during a period after a timer expires, but doing this blocks the computation. In [5], however, processes are nonblocking because the inconsistency was resolved by the information piggybacked in each message. Timer synchronization can also be done using the piggybacked information. But when the transmission delay between two mobile hosts becomes relatively large, the synchronization result will be less accurate.

In this paper, we propose an improved time-based checkpointing protocol that tries to reduce the number of checkpoints. The basic idea is that if a checkpoint initiator does not transitively depend on a process, the process does not have to take a checkpoint associated with the initiator. The result is that the number of checkpoints transmitted over the air can be minimized. Also, and the number of coordinating messages is very small compared to other existing protocols. The protocol is also nonblocking because the inconsistency between processes is avoided by piggybacking necessary information in each message.

The rest of this paper is organized as follows. Section 2 describes the system model. In Section 3 we show the improved timer synchronization technique for time-based protocols. In Section 4 we present our checkpointing protocol and give a performance analysis. Section 5 concludes our work.

Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)
0-7695-1852-4/02 $17.00 © 2002 IEEE

2. System Model and Background

A mobile computing application is executed by a set of N processes running on several mobile hosts (*MH*s). Processes communicate with each other by sending messages. These messages are received and then forwarded to the destination host by the mobile support stations (*MSS*s), which are interconnected by a fixed network. The mobility of *MH*s is supported by Mobile IP, so that messages can be routed to the destination *MH* which is moving around in the network. A *MH* is associated with a *Home Agent* (*HA*)/*Foreign Agent* (*FA*) when it is in the home/foreign network.

To ensure ordered and reliable message deliveries, each message is assigned an increasing sequence number. In the system every process takes a checkpoint periodically. Each checkpoint is associated with a monotonically increasing *checkpoint number*. The time interval after taking the k^{th} checkpoint and before taking the k+1th checkpoint is called the k^{th} *checkpoint interval* (represented as I_k in the following text).

In the system every node (*MH* or *MSS*) contains a system clock, with typical *clock drift rate* ρ in the order of 10^{-5} or 10^{-6}. The system clocks of *MSS*s can be synchronized using Internet time synchronization services such as *Network Time Protocol*, which makes the *maximum deviation* σ of all the clocks within tens of milliseconds. However, in wide area networks, *MSS*s may belong to different organizations. So, we use the clock synchronization protocol to sync the *logical* clocks instead of the *physical* system clocks of *MSS*s. The clocks of *MH*s can be synchronized likewise, but explicit synchronization messages bring overhead to *MH*s because of the limited wireless bandwidth. In addition, the system clocks of *MH*s may not be controlled by a user-level application. Therefore, to coordinate with each other, processes use synchronized *timers* instead of synchronized clocks. The advantages of using timers to coordinate the creation of checkpoints are that the checkpointing protocol does not have to rely on synchronized system clocks, and no explicit synchronization is needed.

Before a mobile computing application starts, a predefined *checkpoint period T* is set on the timers. When the local timer expires, the process saves its system state as a checkpoint. If all the timers expire at exactly the same time, the set of N checkpoints taken at the same instant forms a *globally consistent checkpoint*. Since timers are not perfectly synchronized, the checkpoints may not be consistent because of *orphan* messages. An orphan message m represents an *inconsistent* system state with the event *receive*(m) included in the state while the event *send*(m) not in the state. Orphan messages may lead to *domino effect*, which causes unbounded, cascading rollback propagation. So, by definition, a globally consistent checkpoint is free from the domino effect.

3. Improved Timer Synchronization

In this section we introduce the mechanism of improved timer synchronization. The mechanism then serves as a basis in our checkpointing algorithm, as described in the next section.

The mechanism of timer synchronization in [5] uses piggybacked timer information from the *sender* to adjust the timer at the receiver. When the sender sends a message, it piggybacks its "*time to next checkpoint*" (represented as *timeToCkp*) in the message. The receiver then uses the information to adjust its own *timeToCkp*. The checkpoint number of the sender is also piggybacked in the message, so that the receiver can act accordingly to avoid an orphan message. However, if the timer of the sender is faulty, the erroneous timer information will be spread to the receiver. Besides, since the transmission delay between the sender and the receiver is variable, the timer information from the sender may not reflect the correct situation when the message finally arrives at the receiver.

To achieve more accurate timer synchronization, we utilize the timers in *MSS*s as an absolute reference because timers in the fixed hosts are more reliable than those in *MH*s. We also assume that the timers of the *MSS*s are synchronized every checkpoint period. In our design, *the local MSS of the receiver* is responsible for piggybacking its own *timeToCkp* in every message destined to the receiver, because the *MSS* is the *closest* fixed host to the receiver.

In the system every *MH*/*MSS* maintains a checkpoint number. In the following we use cn_S, cn_D, and cn_{MSS} to represent the checkpoint number of the sender, the receiver, and the local *MSS* of the receiver, respectively. Like [5], the sender piggybacks its own checkpoint number cn_S in each message. When the local *MSS* of the receiver receives the message, apart from *timeToCkp*, it also piggybacks cn_{MSS} in the message, and then it forwards the message to the receiver. So, when receiving the message, the receiver has the following information: cn_S, cn_{MSS}, and *timeToCkp* of the local *MSS* (represented as *m.timeToCkp*). Note that in practice messages take a minimum time td_{min} to be delivered from a *MSS* to a *MH* in its cell. So, whenever the local timer of a *MH* is adjusted by *m.timeToCkp*, substracting td_{min} from *m.timeToCkp* makes the adjustment more accurate. In the following description we use the symbol Δ to represent *minus* td_{min}. The relationship between cn_D, cn_{MSS}, and cn_S determines how the timer is adjusted, as described in the following cases.

I. $\boldsymbol{cn_S = cn_D}$

(1) $cn_{MSS} = cn_S = cn_D$: The receiver resets its *timeToCkp* to "*m.timeToCkp* + Δ".

(2) $cn_{MSS} > cn_S = cn_D$: The timer of MH_D is *late* compared to that of MSS_2. So as soon as message m is processed, MH_D takes a checkpoint with ckpt number cn_{MSS}, and then resets its *timeToCkp* to "*m.timeToCkp* + Δ".

(3) $cn_{MSS} < cn_S = cn_D$: The timers of MH_S and MH_D are both *early* compared to that of MSS_2. MH_D resets its *timeToCkp* to "$T + m.timeToCkp + \Delta$".

II. $cn_S < cn_D$

(1) $cn_S < cn_{MSS} = cn_D$: Since MH_D and its local *MSS* are within the same ckpt period, MH_D just resets its *timeToCkp* to "$m.timeToCkp + \Delta$".

(2) $cn_S = cn_{MSS} < cn_D$: $cn_{MSS} < cn_D$ means that the timer of MH_D expires too early, so MH_D resets its *timeToCkp* to "$T + m.timeToCkp + \Delta$".

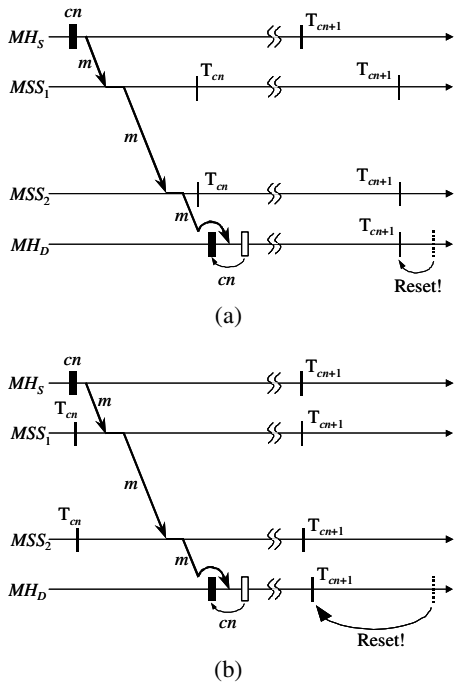

Figure 1. Timer synchronization (a) $cn_S > cn_{MSS} = cn_D$ (b) $cn_S = cn_{MSS} > cn_D$.

III. $cn_S > cn_D$

(1) $cn_S > cn_{MSS} = cn_D$ (Fig. 1(a)): Before MH_D can process *m*, it has to take a ckpt with ckpt number cn_S; otherwise *m* is an orphan message. Then MH_D resets its *timeToCkp* to "$T + m.timeToCkp + \Delta$".

(2) $cn_S = cn_{MSS} > cn_D$ (Fig. 1(b)): MH_D has to take a ckpt before processing *m* in order not to make *m* an orphan message. Since the timer of MH_D is late compared to that of MSS_2 ($cn_{MSS} > cn_D$), MH_D then resets its *timeToCkp* to "$m.timeToCkp + \Delta$".

From the above discussion, we can find that the receiver's timer can be synchronized whenever a message is received. Since the synchronization information is piggybacked in *every* message, the sender's timer can also be synchronized with its local *MSS* as soon as the sender receives the acknowledgement.

In the next section, our checkpointing protocol requires that at the end of a checkpoint interval, none of the *MH*'s timers expires earlier than those of *MSS*s. To fulfill the requirement, we need to take the clock drifts of *MH*s and *MSS*s into account. The clock drift rates of the timers in *MH*s and *MSS*s are represented as ρ_{MH} and ρ_{MSS} respectively. In the system model we also mentioned that after the clock synchronization, there exists a maximum deviation σ between two *MSS*s. In the following lemma, we show how the requirement is achieved.

Lemma 1: *By setting* $\Delta = \sigma + 2\rho_{MSS}\times T + \rho_{MH}\times 2T - td_{min}$ *in the algorithm,* $\forall$*process that has received a message in* I_{cn-1}*, its* I_{cn+1} *begins no earlier than that of a MSS.*

Proof: Assume a process is in I_{cn-1} and it receives a message. It is straightforward that the maximum time deviation between any two *MSS*s after a time period *T*, is $\sigma + 2\rho_{MSS}\times T$. If receiving the message triggers a new ckpt to be taken immediately, the maximum time to the $cn+1^{th}$ ckpt is $2T$. As a result, the maximum time deviation between the process and its *MSS* is $\rho_{MH}\times 2T - td_{min}$ from receiving the message to taking the $cn+1^{th}$ ckpt. By setting $\Delta = \sigma + 2\rho_{MSS}\times T + \rho_{MH}\times 2T - td_{min}$, the adjustment of *timeToCkp* makes the local timer expire no earlier than that of a *MSS* for I_{cn}. On the other hand, if receiving the message does not trigger a new ckpt immediately, the maximum time to the $cn+1^{th}$ ckpt is *T*. But multiplying $2T$ with ρ_{MH} in Δ ensures that even if the process does not receive any message during I_{cn}, the process's I_{cn+1} will not begin earlier than that of a *MSS*. □

4. Time-based Checkpointing Protocol

In this section, we present our time-based checkpointing protocol, which is applicable for mobile computing systems over Mobile IP.

4.1. Notations and Data Structures

- $SoftCkpt^{cn}$: The cn^{th} *soft* checkpoint of a process, saved in the main memory of a *MH*.
- $PermCkpt^{cn}$: The cn^{th} *permanent* checkpoint of a process, saved in the stable storage of the process' *HA* or *FA*. The system recovery line consists of *N* consistent permanent checkpoints, one from each process.
- $Cell_k$: The wireless cell served by MSS_k.
- $Recv_i$: An array of *N* bits of process P_i maintained by P_i's local *MSS*. In the beginning of every checkpoint interval, $Recv_i[j]$ is initialized to 0 for $j = 1$ to *N*, except that $Recv_i[i]=1$. When P_i receives a message *m* from P_j, and the receipt of *m* is confirmed by P_i's *MSS*, $Recv_i[j]$ is set to 1.
- $LastRecv_i$: The $Recv_i$ of the preceding checkpoint interval of process P_i, maintained by P_i's local *MSS*.

- $CkptNum_i$: The current checkpoint number of P_i in the local *MSS*'s knowledge.
- $RejectCP_i$: A variable that saves a checkpoint number of process P_i, maintained by P_i's local *MSS*. When P_i is trying to transmit its soft checkpoint to the *HA*/*FA*, the local *MSS* rejects the transmission if the checkpoint number of the soft checkpoint equals $RejectCP_i$.

4.2. Checkpointing Protocol

4.2.1. Checkpoint initiation. When the local timer expires, a process takes a *soft* checkpoint. More precisely, a soft checkpoint $SoftCkpt^{cn}$ is taken at the beginning of I_{cn}. After a soft checkpoint has been taken, the process resumes its computation.

For simplicity, here we assume that only *one* of the N processes will play the role of the checkpoint initiator of a checkpoint interval. Let's say process P_i decides to act as the initiator of the next checkpoint interval. In the algorithm, P_i has to send a checkpoint request to its local *MSS* during the current checkpoint interval. On receiving the checkpoint request, the *MSS* becomes the *initiator MSS* (denoted by MSS_{init}), which is responsible for collecting and calculating the dependency relationship between the initiator and all other processes in the next checkpoint interval.

4.2.2. Maintaining dependency variables in *MSS*s. Since an *MSS* is responsible for forwarding messages for the processes in its cell, it is reasonable to use the *MSS* to maintain the dependency variables (*Recv*, *LastRecv*) for those processes as well. For example, process P_i in $Cell_k$ receives a message from P_j and then sends an *ACK* back to P_j via MSS_k. By inspecting the *ACK*, MSS_k knows that the message from P_j has been delivered, so MSS_k sets $Recv_i[j]$ to 1. Note that the *ACK* is piggybacked with the checkpoint number of P_i as described in Section 3, which can be used by MSS_k to tell whether P_i has entered the next checkpoint interval or not. As soon as MSS_k finds that P_i has entered a new checkpoint interval, MSS_k saves the current $Recv_i$ as $LastRecv_i$, resets $Recv_i$, and then modifies $Recv_i$ accordingly. At the same time, MSS_k also updates $CkptNum_i$ for P_i. Note that the variable $RejectCP_i$ is also maintained in the *MSS*, but the explanation is left to Section 4.3.2.

4.2.3. Determining the dependency relationship. As soon as the timer of MSS_{init} expires, MSS_{init} broadcasts a *Recv_Request* message to all *MSS*s. At T_{defer} after receiving *Recv_Request*, each *MSS* sends to MSS_{init} the dependency vector (*Recv* or *LastRecv*) of every process in its cell. Here T_{defer} is a tunable parameter that the last message sent by a process before the process's timer expires is expected to arrive at the local *MSS* no later than T_{defer} after the *MSS*'s timer expires. We can choose a proper T_{defer} according to the *QoS* requirements of the wireless network: the better the *QoS*, the smaller the T_{defer}. A reasonable upper bound of T_{defer} can be one half of a checkpoint period ($T/2$), which is normally in the order of several minutes or more.

After receiving all the dependency vectors, MSS_{init} constructs an $N \times N$ dependency matrix D with one row per process. We adopt the algorithm in [6] that by matrix multiplications, all the processes on which the initiator transitively depends can be calculated. In the following we call such processes *initiator-depended processes*. After finishing the calculation, the final dependency vector D_{init} can be obtained, in which $D_{init}[i] = 1$ represents that the initiator transitively depends on P_i in the preceding checkpoint interval.

4.2.4. Discarding unnecessary soft checkpoints. A process can discard the newly taken soft checkpoint if the initiator does not transitively depend on the process in the preceding checkpoint interval. To do that, MSS_{init} obtains a set $S\_Discard^{cn}$ from D_{init}, which consists of any process P_i such that $D_{init}[i] = 0$, and then MSS_{init} sends a notification $DISCARD^{cn}$ to the processes in $S\_Discard^{cn}$. If process P_j receives $DISCARD^{cn}$, it deletes $SoftCkpt^{cn}$ from its main memory, and the local *MSS* of P_j sets $Recv_j = (LastRecv_j \vee Recv_j)$.

On the other hand, if a process does not receive $DISCARD^{cn}$ until T_{decide} after taking $SoftCkpt^{cn}$, it will send $SoftCkpt^{cn}$ to a fixed host to make the checkpoint a *permanent* one. Here T_{decide} is also a tunable parameter, which represents a reasonably long period of time from entering the current checkpoint interval to $DISCARD^{cn}$ should have been delivered to all the processes in $S\_Discard^{cn}$.

4.2.5. Maintaining permanent checkpoints. In order to ensure the robustness of the recovery line, the soft checkpoints in a *MH*'s memory should be transmitted to the stable storage of a fixed host periodically. In a mobile computing system based on Mobile IP, the stable storage of the home agent (*HA*) or foreign agent (*FA*) is an ideal place to store the permanent checkpoints for the processes. When an *HA*/*FA* receives a soft checkpoint $SoftCkpt^{cn}$ from process P_i, it saves $SoftCkpt^{cn}$ in its stable storage as a permanent checkpoint $PermCkpt^{cn}$ of P_i. If $SoftCkpt^{cn}$ of a process is discarded, the process's local *MSS* will inform the process's *HA*/*FA* to renumber $PermCkpt^{cn-1}$ as $PermCkpt^{cn}$ for the process. After the *HA*/*FA* has collected all the checkpoints it should have received, it then proposes to advance the recovery line to checkpoint number *cn*. By adopting any feasible total agreement protocol for distributed systems, the recovery line will be committed to be advanced.

4.2.6. Handling disconnections and handoffs. When an *MH* within its cn^{th} checkpoint interval is about to disconnect with its local *MSS* (say MSS_p), the processes on the *MH* are required to take a soft checkpoint with checkpoint cn+1, and then send these checkpoints to MSS_p. Assume process P_i takes a soft checkpoint $SoftCkpt^{cn+1}$ and sends it to MSS_p. On receiving $SoftCkpt^{cn+1}$, MSS_p saves (i, $SoftCkpt^{cn+1}$) in the stable storage, but MSS_p does not forward $SoftCkpt^{cn+1}$ to P_i's *HA*/*FA* at the moment. The reason is that $SoftCkpt^{cn+1}$ may possibly be discarded later if P_i is in $S\_Discard^{cn+1}$. If MSS_p finds that P_i is not in $S\_Discard^{cn+1}$, it sends $SoftCkpt^{cn+1}$ to P_i's *HA*/*FA* on behalf of P_i. Note that if the *MH* is about to disconnect before T_{decide} after entering the cn^{th} checkpoint interval, P_i has to send $SoftCkpt^{cn}$ along with $SoftCkpt^{cn+1}$ to MSS_p. In this case, MSS_p keeps $SoftCkpt^{cn}$ for P_i until T_{decide} after entering the cn^{th} checkpoint interval: MSS_p may either discard it or send it to P_i's *HA*/*FA*, depending on P_i is in $S\_Discard^{cn}$ or not.

For a disconnected process, its dependency information (*Recv, LastRecv, RejectCP*, *CkptNum*) is still kept in the *MSS*. If the process reconnects with another *MSS* at a later time, the old *MSS* then sends the dependency information of the process to the new *MSS*. For the handoff of a *MH*, the old *MSS* also forwards the dependency information of all the processes in the *MH* to the new *MSS*. If the handoff involves a change of agents, the old agent forwards the permanent checkpoints of the processes in the *MH* to the new agent.

In the following we present a formal description of our checkpointing algorithm:

I. Action at the initiator P_j:

```
01  send Checkpoint_Request to the local MSS;
```

II. Actions at the MSS_{init} when the local timer expires:

```
01  cn ← cn + 1; timeToCkp ← T;
02  send Recv_Request to all MSSs;
03  while (not receiving all Recvs from each MSS)
04      if (timeToCkp = T - T_decide)
05          exit;   /* Abort checkpointing process for this time */
06  construct matrix D;
07  D_init ← calculate(Recv_init, D);  /* Recv_init is Recv of the initiator */
08  S_Discard^cn ← φ;
09  for each P_i:
10      if (D_init[i] = 0)  S_Discard^cn ← S_Discard^cn ∪ P_i;
11  send DISCARD^cn to all processes ∈ S_Discard^cn;
```

III. Actions at process P_i when *Timeout_Event* is triggered for I_{cn}:

```
01  if (SoftCkpt^cn has not been sent to HA/FA)
02      save SoftCkpt^cn in the local disk;
03  take SoftCkpt^(cn+1);
04  cn ← cn + 1; timeToCkp ← nextTimeToCkp;
```

IV. Actions executed at an *MSS*, say MSS_k, in I_{cn}:

```
01  upon relaying message m from P_i ∈ Cell_k to P_j:
02    if (m.cn_i > CkptNum_i)  {
03        CkptNum_i ← m.cn_i; LastRecv_i ← Recv_i; reset Recv_i;
04        modify Recv_i if necessary, then send m to P_j;
05    }
06    else if (m.cn_i = CkptNum_i)
07        modify Recv_i if necessary, then send m to P_j;
08    else   /* m.cn_i < CkptNum_i , m is an out-of-sequence message */
09        send m to P_j;
10  upon receiving Recv_Request from MSS_init:
11    wait (T_defer);
12    for each i that P_i ∈ Cell_k:
13      if (CkptNum_i = cn)   send LastRecv_i to MSS_init;
14      else /* CkptNum_i < cn, and CkptNum_i cannot be larger than cn */ {
15          for any j that a message from P_j is unacknowledged:
16              Recv[j] ← 1;
17          send Recv_i to MSS_init; LastRecv_i ← Recv_i;
18          reset Recv_i; CkptNum_i ← cn;
19      }
20  upon receiving DISCARD^cn for P_i in Cell_k from MSS_init:
21    if (P_i is disconnected)   discard SoftCkpt^cn of P_i;
22    else   forward DISCARD^cn to P_i;
23    Recv_i ← LastRecv_i ∨ Recv_i;
24  upon receiving Disconnect_Request from MH_q in Cell_k:
25    for each P_i in MH_q:  /* SoftCkpt^(cn+1) is included in the request */
26      save SoftCkpt^(cn+1) of P_i in the local disk;
27  upon receiving Handoff_Request from MH_q in Cell_k:
28    for each P_i in MH_q:
29      send (Recv_i, LastRecv_i, CkptNum_i, RejectCP_i) to the
            new MSS of P_i;
30  upon T_decide after entering the cn^th checkpoint interval:
31    for any i such that DISCARD^cn for P_i ∈ Cell_k is undelivered:
32      RejectCP_i ← cn;
33    upon receiving ForwardCP_Request(cn) from P_i ∈ Cell_k:
34      if (RejectCP_i ≠ cn)
35          receive and then forward the ckpt to the HA/FA of P_i;
36      else   reject the transmission;
37  upon expiration of the local timer:
38    cn ← cn + 1; timeToCkp ← T;
```

V. Actions for any process P_i in Icn:

```
01  upon sending SoftCkpt^cn to the HA or FA:
02    send ForwardCP_Request(cn) to the local MSS;
03    if (request not rejected)   send SoftCkpt^cn to the HA or FA;
04  upon receiving DISCARD^cn:
05    discard SoftCkpt^cn;
06  upon expiration of the local timer:
07    nextTimeToCkp ← T; trigger Timeout_Event;
08  upon receiving a message m from P_j:
09    if (m.cn_j = cn) {
10      deliverMsgToProcess(m);
11      if (m.cn_MSS = m.cn_j)   timeToCkp ← m.timeToCkp + Δ;
12      else if (m.cn_MSS > m.cn_j) {
13          cn ← m.cn_MSS; nextTimeToCkp ← m.timeToCkp + Δ;
14          trigger Timeout_Event;   /* A soft ckpt will be taken */
15      }
16      else   /* m.cn_MSS < m.cn_j */
17          timeToCkp ← T + m.timeToCkp + Δ;
18    }
19    else if (m.cn_j < cn) {
20      deliverMsgToProcess(m);
21      if (m.cn_MSS = cn)   timeToCkp ← m.timeToCkp + Δ;
22      else     timeToCkp ← T + m.timeToCkp + Δ;
23    }
24    else   /* m.cn_j > cn */   {
25      if (m.cn_MSS = cn)   nextTimeToCkp ← T + timeToCkp+Δ;
26      else     /* m.cn_MSS = m.cn_j */
27          nextTimeToCkp ← m.timeToCkp + Δ;
28      cn ← m.cn_j;
29      trigger Timeout_Event;   /* A soft ckpt will be taken now */
30      wait until SoftCkpt^cn is taken:
31          deliverMsgToProcess (m);
32    }
```

4.3. Handling Untimely Delayed Messages

In this section we discuss the problem of untimely delayed messages in the network. Since there exists inherent uncertainty of message delivery time in the wired and wireless network, we have to deal with untimely delayed messages in the checkpointing algorithm carefully.

4.3.1. Untimely delayed *Recv* vectors. When MSS_{init} is collecting the *Recv* vectors, it is possible that because of network congestions or link failures in the wired network, some of the *Recv* vectors have not been received until T_{decide} after entering the current checkpoint interval. In this case, the checkpointing process for this time has to be aborted (see code II of the checkpointing algorithm, lines 03-05). In effect, aborting the checkpointing process does not stop the progression of the recovery line since *every* process has taken a soft checkpoint, and these soft checkpoints will become *permanent* when they are sent to the *HA*s or *FA*s.

4.3.2. Untimely delayed $DISCARD^{cn}$ notifications. An inconsistency situation may occur due to untimely delayed $DISCARD^{cn}$ notifications. Although we can choose a proper T_{decide} value such that the untimely delayed notifications are very rare, our algorithm has to cope with the problem in order to ensure the consistency of the global checkpoints. Let's demonstrate the problem as illustrated in Figure 2.

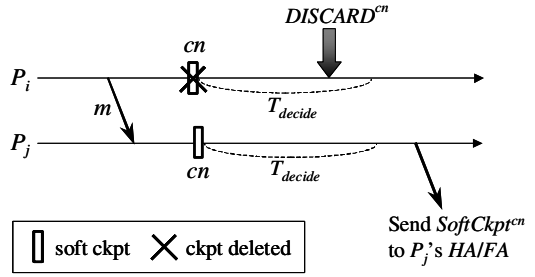

Figure 2. A possible scenario that the delivery of $DISCARD^{cn}$ for P_j is delayed.

Assume P_i and P_j are both in $S\_Discard^{cn}$, but P_j does not receive $DISCARD^{cn}$ until T_{decide} after entering I_{cn}. For P_i, since its $SoftCkpt^{cn}$ is discarded, its *HA*/*FA* will renumber P_i's $PermCkpt^{cn-1}$ as $PermCkpt^{cn}$. For P_j, it will send its $SoftCkpt^{cn}$ to its *HA*/*FA* in order to make the checkpoint permanent. However, if there exists a message m between P_i and P_j, m will be an orphan message with respect to P_i's $PermCkpt^{cn}$ and P_j's $PermCkpt^{cn}$. To cope with the problem, we introduce the variable *RejectCP* of a process, which is also maintained by the local *MSS* of the process. In the above example, the local *MSS* of P_j is aware that $DISCARD^{cn}$ for P_j has not been delivered until T_{decide} after entering I_{cn}, so it sets $RejectCP_j$ to *cn*. Afterwards when P_j tries to send its $SoftCkpt^{cn}$ to *HA*/*FA*, the *MSS* rejects the transmission because $RejectCP_j$ equals *cn* (see code IV, lines 30-36). Therefore, P_j's $PermCkpt^{cn-1}$ will be renumbered as $PermCkpt^{cn}$ so that the inconsistency no longer exists. On P_j's part, if the transmission of its $SoftCkpt^{cn}$ is rejected by the local *MSS*, P_j deletes $SoftCkpt^{cn}$.

4.3.3. Untimely delayed acknowledgements. In our algorithm, the *MSS* maintains the dependency vectors *Recv* and *LastRecv* for a process by inspecting the piggybacked information in an *ACK* sent by the process, but an untimely delayed *ACK* could be a problem during the checkpointing process. Take Figure 3 as an example, when MSS_k is about to send $Recv_i$ to MSS_{init}, *ACK.m* has not arrived so that MSS_k cannot tell whether or not to include the receipt of m in $Recv_i$ at the instant. In our algorithm we take the following policy (refer to code IV, lines 14-19): when MSS_k is about to send $Recv_i$ to MSS_{init} and it finds that such an unacknowledged message exists, $Recv_i[j]$ is set to 1. That is, MSS_k presumes the case in Figure 3(a) always occurs. But if *ACK.m* finally arrives and shows that Figure 3(b) is true instead, the receipt of m is then included in $Recv_i$ of I_{cn}.

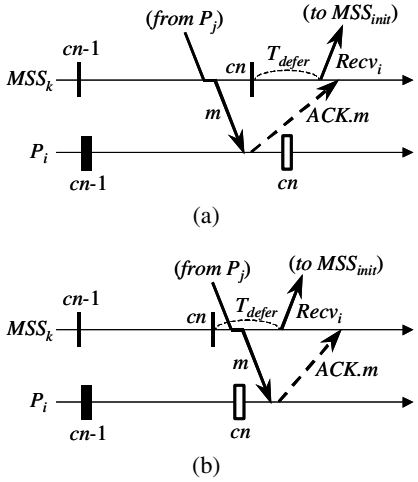

Figure 3. The *ACK* of m arrives later than MSS_k has sent $Recv_i$ to MSS_{init} (a) Receipt of m is in I_{cn-1} of P_i (b) Receipt of m is in I_{cn} of P_i.

4.4. Rollback Recovery

When a failure occurs, all the processes roll back to the latest recovery line. Assume the latest recovery line is numbered as *cn*. For a non-faulty process, if its $SoftCkpt^{cn}$ is still in the main memory and its *RejectCP* is not *cn*, it can roll back to the state of $SoftCkpt^{cn}$ because the content of $SoftCkpt^{cn}$ is identical to $PermCkpt^{cn}$. Otherwise, the process requests its $PermCkpt^{cn}$ from the *HA* or *FA*.

From the above description, we can see that with the help of local soft checkpoints, some of the processes can be recovered locally so that the recovery can be done efficiently.

4.5. Proofs of Correctness

Lemma 2. *If a process P_i receives a message from another process P_j during I_{cn-1} and $P_j \in S\_Discard^{cn}$, then $P_i \in S\_Discard^{cn}$.*

Proof: If $P_i \notin S\_Discard^{cn}$, from the proposed algorithm, the initiator transitively depends on P_i during I_{cn-1}. Since P_i depends on P_j, the initiator also transitively depends on P_j during I_{cn-1}. From the proposed algorithm, $P_j \notin S\_Discard^{cn}$. A contradiction. □

Lemma 3. *N permanent checkpoints with the same checkpoint number form a globally consistent checkpoint.*

Proof: We prove it by induction. In the beginning, the N permanent ckpts with ckpt number 0 obviously form a globally consistent ckpt. Assume there are N permanent ckpts with ckpt number k and they form a globally consistent ckpt. In the proposed algorithm, if a process P_i receives a message m from another process P_j during I_k, there are two possibilities:

Case 1: If $P_j \in S\_Discard^{k+1}$, there are two possibilities for P_j:

- **1.1** P_j does not receive $DISCARD^{k+1}$ until T_{decide} after entering I_{k+1}. From Section 4.3.2 we know P_j's local *MSS* will set $RejectCP_j$ to k+1, so that P_j's $SoftCkpt^{k+1}$ will not be saved as $PermCkpt^{k+1}$. It is P_j's $PermCkpt^k$ be renumbered as $PermCkpt^{k+1}$.
- **1.2** P_j receives $DISCARD^{k+1}$ before T_{decide} after entering I_{k+1}. In this case, P_j discards $SoftCkpt^{k+1}$ and the preceding permanent ckpt $PermCkpt^k$ of P_j is renumbered as $PermCkpt^{k+1}$.

From Lemma 2 we know $P_i \in S\_Discard^{k+1}$. Through the above discussion, we know no matter if P_i receives $DISCARD^{k+1}$ or not, the preceding permanent ckpt $PermCkpt^k$ of P_i is renumbered as $PermCkpt^{k+1}$. Since the permanent ckpts with ckpt number k form a globally consistent ckpt, there is no orphan message between the k+1th permanent ckpt of P_i and the k+1th permanent ckpt of P_j.

Case 2: If $P_j \notin S\_Discard^{k+1}$, P_j does not receive $DISCARD^{k+1}$ and its $SoftCkpt^{k+1}$ is sent to *HA*/*FA* and saved as $PermCkpt^{k+1}$. From the proposed algorithm, P_j must send m before it takes $SoftCkpt^{k+1}$. Otherwise, P_i will take $SoftCkpt^{k+1}$ before processing m, which makes m been received within P_i's I_{k+1}. As a result, no matter P_i's $PermCkpt^k$ is renumbered as $PermCkpt^{k+1}$ or P_i's $SoftCkpt^{k+1}$ is saved as $PermCkpt^{k+1}$, there is no orphan message between P_i's $PermCkpt^{k+1}$ and P_j's $PermCkpt^{k+1}$.

Thus, if the N permanent checkpoints with ckpt number k form a globally consistent ckpt, there is no orphan message between the k+1th permanent ckpts of any two processes. That is, N permanent ckpts with ckpt number k+1 form a globally consistent ckpt. □

Theorem. *The proposed algorithm always creates a consistent global checkpoint.*

Proof: In the beginning there are N permanent ckpts with ckpt number 0, and they form the initial recovery line. Suppose there exists N permanent ckpts with the same ckpt number k. In the proposed algorithm, we advance the recovery line to ckpt number k+1 only when all processes' permanent ckpts $PermCkpt^{k+1}$ are collected. From Lemma 3, N permanent ckpts with the same ckpt number form a globally consistent ckpt. Therefore, there always exists a consistent global ckpt. □

4.6. Performance Analysis

In this section we discuss the performance of our checkpointing algorithm, including the blocking time, the number of permanent checkpoints, and the number of coordinating messages. Then we show the comparison with other protocols in a table. Here are the notations used in the following text:

- N_{min}: the number of processes that need to take checkpoints using the Koo-Toueg algorithm [1].
- N_{dep}: the average number of processes on which a process depends. ($1 \le N_{dep} \le N - 1$)
- $C_{wireless}$: cost of sending a message in the wireless link.
- C_{wired}: cost of sending a message in the wired link.
- C_{broad}: cost of broadcasting a message to all processes.
- T_{ckpt}: the checkpointing time, including the delays incurred in transferring a checkpoint from a *MH* to its *MSS* and saving the checkpoint in the stable storage in the *MSS* or a fixed host.

4.6.1. Blocking time. It is very clear that the blocking time of our protocol is 0.

4.6.2. Number of new permanent checkpoints. In Section 4.3.3, we described that if there is an unacknowledged message like the scenario depicted in Figure 3, the *MSS* presumes the case in Figure 3(a) always occurs. That is, the receipt of message m from P_j is included in the *Recv* vector of P_i's I_{cn-1}. If it turns out later that Figure 3(b) is true instead, then there is a chance that P_j and P_j-depended processes should not have been included in the dependency with the initiator. The consequence is that there may be additional soft checkpoints been made permanent, so as to increase the number of new permanent checkpoints. If we choose a proper T_{defer} such that the untimely delayed *ACK*s are very rare, the number of new permanent checkpoints is then close to minimum.

4.6.3. Number of coordinating messages. In the algorithm, the only coordinating message transmitted in the wireless link is the discard notification to a process in the set $S\_Discard^{cn}$. The approximate number of discard

notifications is $N - N_{min}$. Messages sent in the wired link are N *Recv* vectors from *MSS*s to MSS_{init}, and $N - N_{min}$ discard notifications from MSS_{init} to *MSS*s that serve the processes in $S\_Discard^{cn}$.

4.6.4. Comparison with other algorithms. Table 1 compares the performance of our algorithm with the algorithms in [1], [10], [12]. Compared to the Neves-Fuchs algorithm which is also time-based, our algorithm reduces the number of checkpoints to nearly minimum, so that the total number of checkpoints transmitted onto the fixed network is reduced. Fewer checkpoints transmitted also means less power consumption for mobile hosts. For a mobile computing system, it is also very critical to minimize the number and size of the messages transmitted in the wireless link. So, if we only consider the number of coordinating messages sent in the wireless link, our algorithm performs fairly well. For the size of the piggybacked information and the coordination message in the wireless link, our protocol outperforms Cao-Singhal algorithm with $O(1)$ to $O(N)$. On the other hand, the cost of transmitting a message in the wired link is far less than transmitting in the wireless link. So, although our protocol requires $O(N)$ coordinating messages in the wired network, the cost is affordable for wired networks with high bandwidth.

Table 1. Performance Comparison*

Algorithm	Blocking time	# of ckpts	# of messages
Koo-Toueg [1]	$N_{min} \times T_{ckpt}$	N_{min}	$3\times N_{min}\times N_{dep}\times (C_{wired}+C_{wireless})$
Neves-Fuchs [12]	$\sigma + 2\rho_{MH}T - td_{min}$	N	$2\times N\times C_{wireless}$
Cao-Singhal [10]	0	N_{min}	$\approx 2\times N_{min}\times (C_{wired}+C_{wireless}) + min(N_{min}\times(C_{wired}+C_{wireless}),C_{broad})$
Our algorithm	0	$\approx N_{min}$	$\approx (N-N_{min}) \times (C_{wired}+C_{wireless}) + N \times C_{wired}$

* The performance data of algorithms [1] and [10] are from [10].

5. Conclusions

In this paper we have proposed a time-based checkpointing protocol for mobile computing systems over Mobile IP. Our protocol reduces the number of checkpoints compared to the traditional time-based protocols. We also make use of the accurate timers in the *MSS*s to adjust the timers in the *MH*s, so that our protocol is well suited to mobile computing systems with *MH*s spread across a wide area network. We also take advantage of the infrastructure provided by Mobile IP, so that the permanent checkpoints of the participating processes can be saved in the *HA* or *FA* depending on the process's current location. Compared to other protocols, our protocol performs very well in the aspects of minimizing the number and size of messages transmitted in the wireless media. Tracking and computing the dependency relationship between processes are performed in the *MSS*s, so that *MH*s are free from additional tasks during checkpointing.

6. Acknowledgement

This research was supported in part by the Development of Communication Software Core Technology project of Institute for Information Industry and sponsored by MOEA, R.O.C.

References

[1] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, pp. 23-31, Jan. 1987.

[2] Z. Tong, R. Y. Kain, and W. T. Tsai, "A Low Overhead Checkpointing and Rollback Recovery Scheme for Distributed Systems," *Proc. of the 8th Symp. on Reliable Distributed Systems*, pp. 12-20, Oct. 1989.

[3] A. Acharya and B. R. Badrinath, "Checkpointing Distributed Applications on Mobile Computers," *Proc. of Int'l Conf. on Parallel and Distributed Information Systems*, pp. 73-80, Sep. 1994.

[4] R. Prakash and M. Singhal, "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7(10), pp. 1035-1048, Oct. 1996.

[5] N. Neves and W. K. Fuchs, "Adaptive Recovery for Mobile Environments," *Comm. of the ACM*, pp. 68-74, Jan. 1997.

[6] G. Cao and M. Singhal, "On the Impossibility of Min-Process Non-Blocking Checkpointing and An Efficient Checkpointing Algorithm for Mobile Computing Systems," *Proc. of the 27th Int'l Conf. on Parallel Processing*, pp. 37-44, Aug. 1998.

[7] H. Higaki and M. Takizawa, "Checkpoint-Recovery Protocol for Reliable Mobile Systems," *Proc. of the IEEE Symp. on Reliable Distributed Systems*, pages 93-99, Oct. 1998.

[8] K. F. Ssu, B. Yao, W. K. Fuchs, N. Neves, "Adaptive Checkpointing with Storage Management for Mobile Environments," *IEEE Trans. on Reliability*, Vol. 48(4), pp. 315-324, Dec. 1999.

[9] T. Park and H. Y. Yeom, "An Asynchronous Recovery Scheme based on Optimistic Message Logging for Mobile Computing Systems," *Proc. of the Int'l Conf. on Distributed Computing Systems*, pp. 436-443, Apr. 2000.

[10] G. Cao and M. Singhal, "Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 12(2), pp. 157-172, Feb. 2001.

[11] T. Park, N. Woo, and H. Y. Yeom, "An Efficient Recovery Scheme for Mobile Computing Environments," *IEEE Int'l Conf. on Parallel and Distributed Systems*, Jun. 2001.

[12] N. Neves and W. K. Fuchs, "Coordinated Checkpointing Without Direct Coordination," *Proc. of the IEEE Int'l Computer Performance & Dependability Symp.*, pp. 23-31, Sep. 1998.