Mobile Networks and Applications 8, 687–697, 2003

An Efficient Time-Based Checkpointing Protocol for Mobile Computing Systems over Mobile IP

CHI-YI LIN, SZU-CHI WANG and SY-YEN KUO *
Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

Abstract. Time-based coordinated checkpointing protocols are well suited for mobile computing systems because no explicit coordination message is needed while the advantages of coordinated checkpointing are kept. However, without coordination, every process has to take a checkpoint during a checkpointing process. In this paper, an efficient time-based coordinated checkpointing protocol for mobile computing systems over Mobile IP is proposed. The protocol reduces the number of checkpoints per checkpointing process to nearly minimum, so that fewer checkpoints need to be transmitted through the costly wireless link. Our protocol also performs very well in the aspects of minimizing the number and size of messages transmitted in the wireless network. In addition, the protocol is nonblocking because inconsistencies can be avoided by the piggybacked information in every message. Therefore, the protocol brings very little overhead to a mobile host with limited resource. Additionally, by taking advantage of reliable timers in mobile support stations, the time-based checkpointing protocol can adapt to wide area networks.

Keywords: mobile computing, fault tolerance, checkpointing and rollback-recovery

Category: Fault tolerance for IPv6 or IPv4-based mobile computing

1. Introduction

As computer and wireless communication technology advance, the paradigm of mobile computing becomes close to reality. Mobile users are able to access and exchange information while they are on the move. As a result, collaborative work can be done easily, no matter where the participating members are physically located. Moreover, since user mobility is supported, it is possible that joint workers are distributed over the wide area network and each of them is connected to the network via a wireless link. For example, in a sensor network which carries out a real-time scientific computation, sensors with processing capability can be mobile and distributed. Traveling salespersons may rely on the gradation of each other for making an appropriate decision at a particular time. In these scenarios, the most important thing is to make sure their work is progressing, and to minimize the lost work if a failure occurs.

To achieve high reliability, checkpointing and rollback-recovery techniques are widely used in the parallel and distributed computing environment [4–6,8,18]. Recently, checkpointing protocols for mobile computing systems have also been proposed [1–3,7,9,12,14–17]. A common goal of checkpointing protocols for mobile environment is to avoid extra coordination messages and unnecessary checkpoints. Coordinated checkpointing protocols have the advantage of simplicity over uncoordinated checkpointing protocols in terms of recovery and garbage collection. Besides, the former requires less storage capacity for saving checkpoints. Traditional coordinated checkpointing protocols synchronize processes by exchanging coordination messages, but these coordination messages can be avoided by using *time-based* protocols [4,10–13,18], which use synchronized clocks or timers to indirectly coordinate the creation of checkpoints.

However, time-based protocols require *every* process to take a checkpoint during a checkpointing process, which is saved in the local disk of a mobile host or sent to a fixed host. In this paper, we propose an efficient 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. So, in our protocol, every process takes a *soft* checkpoint first, and then the soft checkpoint will be discarded when the process is found to be irrelevant to the initiator. Soft checkpoints are saved in the main memory of mobile hosts, and the soft checkpoints will be saved in the local disk at a later time only if they are not discarded. As a result, the number of disk accesses in the mobile hosts can be reduced.

Our protocol achieves the performance that the number of checkpoints is close to minimum, and the number of coordinating messages is very small compared to other existing protocols. The protocol is also non-blocking because the inconsistency between processes is avoided with the aid of the piggybacked information in each message.

The rest of this paper is organized as follows. Section 2 briefly introduces the related work. Section 3 describes the system model and the background. In section 4 we show how the timer synchronization can be improved for time-based protocols. In section 5 we present our checkpointing protocol based on the synchronization technique in section 4. Finally, section 6 concludes our work.

* Corresponding author. E-mail: sykuo@cc.ee.ntu.edu.tw

2. Related works

Prakash and Singhal [16] first proposed a checkpointing protocol for mobile computing systems that requires only a minimum number of processes to take checkpoints (called *min-process* property) and does not block the underlying computation (called *non-blocking* property) during checkpointing. However, Cao and Singhal [2] proved that a min-process nonblocking checkpointing algorithm does not exist. They also introduced the concept of *mutable* checkpoint [3], which can be saved in the main memory or local disk of a mobile host, and is neither a tentative nor a permanent checkpoint. In their checkpointing algorithm, the number of new tentative or permanent checkpoints is minimal, excluding those mutable checkpoints that are discarded after a checkpointing process.

Time-based protocols alleviate the need for explicit coordination message when taking a global checkpoint. Since timers cannot be perfectly synchronized, the consistency between all the checkpoints can still be a problem. In [11,13], the consistency problem is solved by disallowing message sending during a period after a timer expires, but this makes a checkpointing protocol become a blocking protocol. In [12], however, processes are nonblocking because the consistency problem 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. Since the reliability of a mobile support station is much higher than that of a mobile host, we can take advantage of the accurate clock or timer in a mobile support station as a reliable reference for *MH*s.

3. 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 mobile support stations (*MSS*s), which are interconnected by a fixed network. The mobility of *MH*s is supported by Mobile IP of IPv4 or IPv6, 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*) when it is within its home network, and with a *Foreign Agent* (*FA*) when within a foreign network.

To ensure ordered and reliable message deliveries, each message is associated with an increasing sequence number. To provide a computing system with fault tolerance capability, every process takes a checkpoint periodically. The periodicity of taking two consecutive checkpoints is called a *checkpoint period*. Each checkpoint is associated with a monotonically increasing *checkpoint number*. In a coordinated checkpointing protocol, those checkpoints with the same checkpoint number from all the processes form a globally consistent checkpoint. The time interval after taking the kth checkpoint and before taking the $(k+1)$th checkpoints is called the kth *checkpoint interval*.

In a system, each node (including *MH*s and *MSS*s) 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* (*NTP*), which makes the *maximum deviation* σ of all the clocks within tens of milliseconds. The synchronization of *MSS*s can be done periodically. However, in a wide area network environment, *MSS*s may belong to different organizations. So, instead of syncing the *physical* system clocks of *MSS*s, we use the clock synchronization protocol to sync the *logical* clocks. 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. It is also possible that there are other applications running concurrently in the *MH*, and some of the applications require transactions. The consequence is that modifying system clocks to coordinate between processes may not be a feasible way. 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 of the participating hosts, 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 system state while the event *send*(m) not in the state. Orphan messages may lead to the *domino effect*, which causes unbounded, cascading rollback propagation. As a result, in designing a checkpointing protocol, we need to ensure that no orphan message exists in a global checkpoint, so that the recovery can be free from the domino effect.

4. Improved timer synchronization

In this section we modify the timer synchronization mechanism in the adaptive checkpointing protocol proposed by Neves and Fuchs [12], which makes the mechanism adaptive to wide area networks.

The mechanism of timer synchronization in [12] 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 piggybacked *timeToCkp* to adjust its own *timeToCkp*. The checkpoint number of the sender is also piggybacked in the

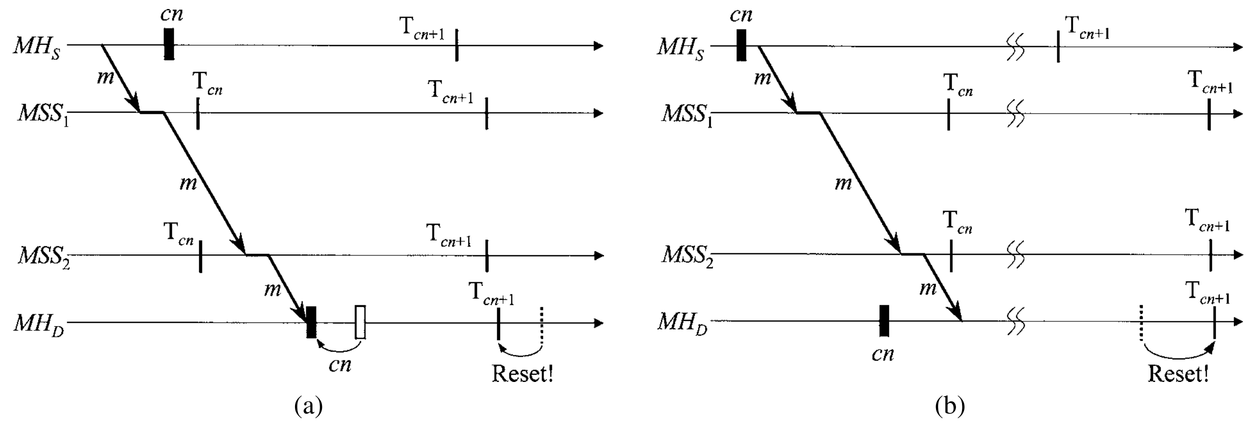


Figure 1. Timer synchronization (a) $cn_{MSS} > cn_S = cn_D$, and (b) $cn_{MSS} < cn_S = cn_D$.

message, so that the receiver can act accordingly to avoid an orphan message. However, it is clear that 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 can utilize the timers in *MSS*s as an absolute reference because these timers in fixed hosts are more reliable than those in *MH*s. We assume that the timers of the *MSS*s are synchronized every checkpoint period, which is not a problem for fixed hosts in a wired network. 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* and *MSS* maintains a checkpoint number. The checkpoint number is incremented whenever the local timer expires. In the following we use cn_S, cn_D, and cn_{MSS} to represent the checkpoint number of the sender, receiver, and the local *MSS* of the receiver, respectively. Just like in [12], 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. So there are three pieces of information piggybacked in a message when the message arrives at the receiver: 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 the checkpoint numbers of the receiver, the cn_D, cn_{MSS}, and cn_S determines how the timer is adjusted, as described in the following cases:

I. Checkpoint numbers of the sender and the receiver are the same ($cn_S = cn_D$):

(1) cn_{MSS} is equal to cn_S and cn_D ($cn_{MSS} = cn_S = cn_D$). In this case, the receiver resets its *timeToCkp* to *m.timeToCkp* $+ \Delta$.

(2) cn_{MSS} is larger than cn_S and cn_D ($cn_{MSS} > cn_S = cn_D$, see figure 1(a)). It means that 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 checkpoint number equals to cn_{MSS}. Then MH_D resets its *timeToCkp* to *m.timeToCkp* $+ \Delta$.

(3) cn_{MSS} is smaller than cn_S and cn_D ($cn_{MSS} < cn_S = cn_D$, see figure 1(b)). It means that the timers of MH_S and MH_D are both *early* compared to that of MSS_2. In order to make the timer of MH_D expire at around the right time, MH_D resets its *timeToCkp* to checkpoint period T plus *m.timeToCkp* $+ \Delta$.

II. Checkpoint number of the sender is smaller than that of the receiver ($cn_S < cn_D$):

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

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

III. Checkpoint number of the sender is larger than that of the receiver ($cn_S > cn_D$):

(1) cn_{MSS} is equal to cn_D ($cn_S > cn_{MSS} = cn_D$, see figure 2(a)). In this case, before MH_D can process m, it has to take a checkpoint with checkpoint number equal to cn_S; otherwise m becomes an orphan message. Then MH_D resets its *timeToCkp* to checkpoint period T plus *m.timeToCkp* $+ \Delta$.

(2) cn_{MSS} is equal to cn_S ($cn_S = cn_{MSS} > cn_D$, see figure 2(b)). This case is basically the same as the previous one: MH_D has to take a checkpoint 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* $+ \Delta$.

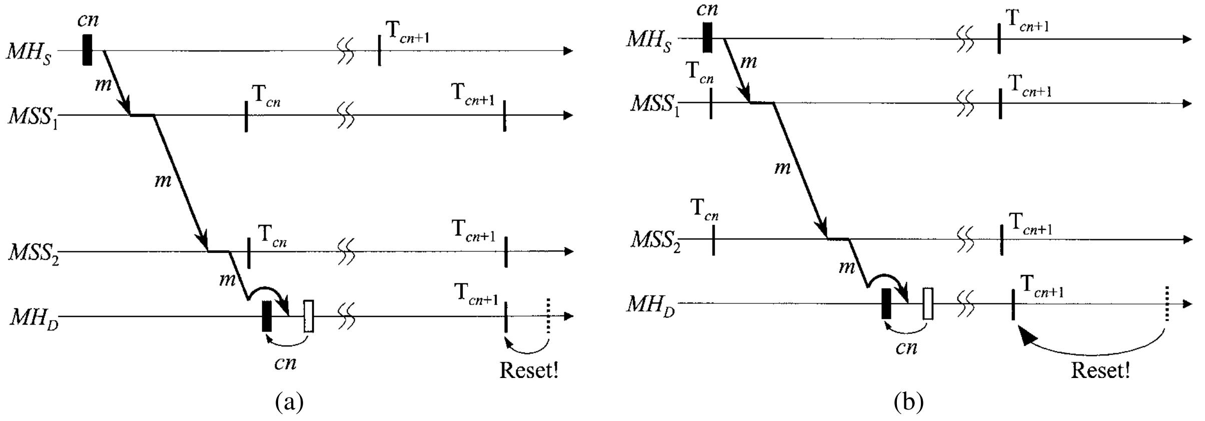


Figure 2. Timer synchronization (a) $cn_S > cn_{MSS} = cn_D$, and (b) $cn_S = cn_{MSS} > cn_D$.

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 other words, both timers of the sender and the receiver can be synchronized in every message exchange.

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, for any process that has received a message in the $(cn - 1)$th checkpoint interval, its $(cn + 1)$th checkpoint interval begins no earlier than that of a MSS.

Proof. Assume a process is in the $(cn - 1)$th checkpoint interval 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 checkpoint to be taken immediately such as cases I(2), III(1), and III(2), the maximum time to the $(cn + 1)$th checkpoint 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 checkpoint. 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 the *cn*th checkpoint interval. On the other hand, if receiving the message does not trigger a new checkpoint immediately such as all other cases, the maximum time to the $(cn + 1)$th checkpoint is T. But multiplying $2T$ with ρ_{MH} in Δ ensures that even if the process does not receive any message during the *cn*th checkpoint interval, the process's $(cn + 1)$th checkpoint interval will not begin earlier than that of a *MSS*. □

5. A time-based coordinated checkpointing protocol

In this section, we present our time-based checkpointing protocol based on the synchronization techniques described in section 4, which is applicable for mobile computing systems over Mobile IP.

5.1. Notations and data structures

- $SoftCkpt^{cn}$: The *cn*th *soft* checkpoint of a process, which is saved in the main memory of a *MH*.
- $HardCkpt^{cn}$: The *cn*th *hard* checkpoint of a process, which is saved in the local disk of a *MH* or the stable storage of the *MH*'s *HA* or *FA*.
- $PermCkpt^{cn}$: The *cn*th *permanent* checkpoint of a process, which is saved in the stable storage of the *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 $Recv_i[i]$ always equals 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 $Recv_i$ vector of the preceding checkpoint interval of process P_i, which is also maintained by P_i's local *MSS*.
- $RejectCP_i$: A variable that saves a checkpoint number for process P_i, maintained by P_i's local *MSS*. When P_i is trying to transmit its hard checkpoint to the wired network, the local *MSS* rejects the transmission if the checkpoint number equals $RejectCP_i$.
- $CkptNum_i$: The current checkpoint number of P_i in the local *MSS*'s knowledge.

5.2. The checkpointing protocol

5.2.1. Checkpoint initiation

A process takes a *soft* checkpoint whenever its local timer expires. That is, the checkpointing process is intrinsically initiated distributedly, by the event of timer expiration in each mobile host. From section 4 we know that a process may also be forced to take a soft checkpoint on receiving a message. In this case, *timeToCkp* of the process is effectively set to zero on receiving the message, so a soft checkpoint will be taken immediately. After a soft checkpoint has been taken, P_i resumes its computation.

We assume that during each checkpointing process, one of the N processes will play the role of the checkpoint initiator. During a checkpointing interval, the initiator sends a checkpoint request *only* to its local *MSS* (denoted by MSS_{init}). During the checkpointing process, MSS_{init} is then responsible for collecting and calculating the dependency relationship between the initiator and all other processes. Note that if there is no checkpoint initiator, the protocol becomes a traditional time-based checkpointing protocol, which makes no effort to reduce unnecessary checkpoints.

5.2.2. Maintaining dependency variables in MSSs

Since a *MSS* is responsible for forwarding messages to/from the *MH*s in its cell, we can make use of the *MSS* to maintain the dependency variables (*Recv*, *LastRecv*) for those processes in the cell. 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 4. From the piggybacked checkpoint number of P_i, MSS_k can 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 then 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* is also maintained in the *MSS*. We will explain it in section 5.3.2.

5.2.3. Calculating the dependency relationship

When the local 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* of the mobile 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 [2] that by matrix multiplications, all the processes on which the initiator transitively depends can be calculated. 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.

5.2.4. Discarding unnecessary soft checkpoints

A process can discard the newly taken soft checkpoint if the initiator does not 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_i receives $DISCARD^{cn}$, it deletes the soft checkpoint from its main memory, and then renumbers the old $HardCkpt^{cn-1}$ as $HardCkpt^{cn}$. For the *MSS* that delivers $DISCARD^{cn}$ to P_i, it sets $Recv_i = (LastRecv_i \vee Recv_i)$ for P_i.

On the other hand, for those processes that do not receive $DISCARD^{cn}$ during the *cn*th checkpoint interval, the soft checkpoints are kept in the main memory. At the beginning of the $(cn+1)$th checkpoint interval, the soft checkpoint is turned into a *hard* one before a new soft checkpoint can be taken.

5.2.5. Maintaining permanent checkpoints

In order to ensure the robustness of the recovery line, the hard checkpoints in a *MH*'s local disk should be transmitted to the stable storage of a fixed host periodically. In the 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 in a *MH*.

The initial state of a process can be regarded as the first permanent checkpoint saved in the *HA* or *FA* of the process. So, there exists an initial recovery line which consists of N permanent checkpoints with checkpoint number 0. From 5.2.4 we know that a set of N hard checkpoints with checkpoint number *cn* will be formed at the beginning of the $(cn + 1)$th checkpoint interval. During the $(cn+1)$th checkpoint interval, each process sends its $HardCkpt^{cn}$ to either the *HA* or *FA*, depending on its current location. When the *HA*/*FA* receives the checkpoint, it saves $HardCkpt^{cn}$ in its stable storage as a permanent checkpoint $PermCkpt^{cn}$. 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, when all the *HA*s and *FA*s have proposed, the recovery line is advanced to checkpoint number *cn*.

5.2.6. Handling disconnections and Handoffs

When a *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 hard checkpoint with checkpoint $cn + 1$. Then, the checkpoints are forwarded to the *HA* or *FA* by MSS_p. Assume process P_i takes a hard checkpoint $HardCkpt^{cn+1}$. On receiving $HardCkpt^{cn+1}$, the *HA/FA* saves $(i, HardCkpt^{cn+1})$ in the stable storage, but $HardCkpt^{cn+1}$ does not overwrite $PermCkpt^{cn+1}$ of P_i at the moment. The reason is that $HardCkpt^{cn+1}$ may possibly be discarded during the $(cn + 1)$th checkpoint interval if P_i is in $S\_Discard^{cn+1}$. If $HardCkpt^{cn+1}$ is not discarded during the $(cn+1)$th checkpoint interval, it is turned into a permanent checkpoint during the $(cn + 2)$th checkpoint interval.

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:

```
send $Checkpoint\_Request$ to the local $MSS$; /* The $MSS$ becomes $MSS_{init}$ */
```

II. Actions at the MSS_{init}:

```
wait until the local timer expires; /* Enters a new checkpoint interval */
$cn \leftarrow cn + 1$; $timeToCkp \leftarrow T$;
send $Recv\_Request$ to all $MSS$s;
while (not receiving all $Recv$ vectors from each $MSS$)
    if ($timeToCkp = 0$)
        exit; /* Abort checkpointing process for this time */
construct matrix $D$;
$D_{init} \leftarrow$ calculate($Recv_{init}$, $D$); /* $Recv_{init}$ is the $Recv$ variable of the initiator */
$S\_Discard^{cn} \leftarrow \phi$;
for each $P_i$:
    if ($D_{init}[\,i\,] = 0$ )
        $S\_Discard^{cn} \leftarrow S\_Discard^{cn} \cup P_i$;
send $DISCARD^{cn}$ to all processes $\in S\_Discard^{cn}$;
```

III. Actions at process P_i when *Timeout_Event* is triggered for checkpoint interval *cn*:

```
if ($SoftCkpt^{cn} \neq$ NULL)
    turn $SoftCkpt^{cn}$ into $HardCkpt^{cn}$;
take $SoftCkpt^{cn+1}$;
$cn \leftarrow cn + 1$; $timeToCkp \leftarrow nextTimeToCkp$;
```

IV. Actions executed at an *MSS*, say MSS_k, in the checkpoint interval *cn*:

```
upon relaying message $m$ from $P_i \in Cell_k$ to $P_j$: /* Note that $m$ can also be an $ACK$ */
    if ($m.cn_i > CkptNum_i$) /* $P_i$ has entered next ckpt interval but $MSS_k$ is not aware of that*/
    {
        $CkptNum_i \leftarrow m.cn_i$;
        $LastRecv_i \leftarrow Recv_i$;
        reset $Recv_i$;
        modify $Recv_i$ if necessary, then send $m$ to $P_j$;
    }
    else if ($m.n_i = CkptNum_i$)
        modify $Recv_i$ if necessary, then send $m$ to $P_j$;
    else /* $m.cn_i < CkptNum_i$ */
        send $m$ to $P_j$;
upon receiving $Recv\_Request$ from $MSS_{init}$:
    wait ($T_{defer}$);
    for each $i$ that $P_i \in Cell_k$:
        if ($CkptNum_i = cn$)
            send $LastRecv_i$ to $MSS_{init}$;
        else /* $CkptNum_i < cn$, and $CkptNum_i$ cannot be larger than $cn$ */
        {
            for any $j$ that a message from $P_j$ is unacknowledged:
                $Recv_i[j] \leftarrow 1$;
            send $Recv_i$ to $MSS_{init}$;
            $LastRecv_i \leftarrow Recv_i$; reset $Recv_i$;
            $CkptNum_i \leftarrow cn$;
        }
upon receiving $DISCARD^{cn}$ for $P_i$ in $Cell_k$ from $MSS_{init}$:
    if ($P_i$ is disconnected)
        forward $DISCARD^{cn}$ to the $HA/FA$ of $P_i$;
    else
        forward $DISCARD^{cn}$ to $P_i$;
    $Recv_i \leftarrow LastRecv_i \vee Recv_i$;
upon receiving $Disconnect\_Request$ from $MH_q$ in $Cell_k$:
    for each $P_i$ in $MH_q$: /* $HardCkpt^{cn+1}$ is included in the request */
        send $HardCkpt^{cn+1}$ to the $HA/FA$ of $P_i$;
upon receiving $Handoff\_Request$ from $MH_q$ in $Cell_k$:
    for each $P_i$ in $MH_q$: /* The id of the new $MSS$ is included in the request */
        send ($Recv_i$, $LastRecv_i$, $RejectCP_i$, $CkptNum_i$) to the new $MSS$ of $P_i$;
upon the local timer expires:
```

```
39      for any $i$ such that $DISCARD^{cn}$ for $P_i \in Cell_k$ is undelivered:
40          $RejectCP_i \leftarrow cn$;
41      $cn \leftarrow cn + 1$; $timeToCkp \leftarrow nextTimeToCkp$;
42  upon receiving $ForwardCP\_Request(cn-1)$ from $P_i \in Cell_k$:
43      if ($cn-1 \neq RejectCP_i$)
44          receive and then forward the checkpoint to the $HA/FA$ of $P_i$;
45      else
46          reject the transmission;
```

V. Actions for any process P_i in the checkpoint interval *cn*:

```
01  upon sending $HardCkpt^{cn-1}$ to the $HA$ or $FA$:
02      send $ForwardCP\_Request(cn-1)$ to the local $MSS$;
03      if (request not rejected)
04          send $HardCkpt^{cn-1}$ to the $HA$ or $FA$;
05  upon receiving $DISCARD^{cn}$:
06      discard $SoftCkpt^{cn}$; renumber $HardCkpt^{cn-1}$ as $HardCkpt^{cn}$;
07  upon expiration of the local timer:
08      $nextTimeToCkp \leftarrow T$; trigger $Timeout\_Event$;
09  upon receiving a message $m$ from $P_j$:
10      if ($m.cn_j = cn$)
11      {
12          deliverMsgToProcess($m$);
13          if ($m.cn_{MSS} = m.cn_j$)
14              $timeToCkp \leftarrow m.timeToCkp + \Delta$;
15          else if ($m.cn_{MSS} > m.cn_j$)
16          {
17              $cn \leftarrow m.cn_{MSS}$; $nextTimeToCkp \leftarrow m.timeToCkp + \Delta$;
18              trigger $Timeout\_Event$; /* A soft ckpt will be taken now */
19          }
20          else
21              $timeToCkp \leftarrow T + m.timeToCkp + \Delta$; /* $m.cn_{MSS} < m.cn_j$ */
22      }
23      else if ($m.cn_j < cn$)
24      {
25          deliverMsgToProcess($m$);
26          if ($m.cn_{MSS} = cn$)
27              $timeToCkp \leftarrow m.timeToCkp + \Delta$;
28          else
29              $timeToCkp \leftarrow T + m.timeToCkp + \Delta$; /* $m.cn_{MSS} = m.cn_j$ */
30      }
31      else /* $m.cn_j > cn$ */
32      {
33          if ($m.cn_{MSS} = cn$)
34              $nextTimeToCkp \leftarrow T + timeToCkp + \Delta$;
35          else
36              $nextTimeToCkp \leftarrow m.timeToCkp + \Delta$; /* $m.cn_{MSS} = m.cn_j$ */
37          $cn \leftarrow m.cn_j$;
38          trigger $Timeout\_Event$; /* A soft ckpt will be taken now */
39          wait until $SoftCkpt^{cn}$ is taken:
40              deliverMsgToProcess ($m$);
41      }
```

5.3. Handling untimely delayed messages

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

5.3.1. 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, at the end of the checkpoint interval, some of the *Recv* vectors have not been received yet. In this case, the checkpointing process for this time has to be aborted (see code II, lines 04–06). 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 be turned *hard* at the beginning of the next checkpoint interval.

5.3.2. Untimely delayed $DISCARD^{cn}$ notifications

There is a chance that the discard notifications $DISCARD^{cn}$ cannot be delivered to some of the processes before their *cn*th checkpoint interval is over. For example, the scenario in figure 3 can happen for any P_i and P_j both in $S\_Discard^{cn}$: at the end of the *cn*th checkpoint interval, P_i has received $DISCARD^{cn}$ but P_j has not. The consequence is that P_i will renumber its $HardCkpt^{cn-1}$ as $HardCkpt^{cn}$ and P_j will turn its $SoftCkpt^{cn}$ into $HardCkpt^{cn}$. If there exists a message m

sent by P_i after P_i's $HardCkpt^{cn-1}$ and received by P_j before P_j's $SoftCkpt^{cn}$, then m becomes an orphan message with respect to P_i's $HardCkpt^{cn}$ and P_j's $HardCkpt^{cn}$. However, the local MSS of P_j is aware that P_j does not receive $DISCARD^{cn}$ during the cnth checkpoint interval, so it sets $RejectCP_j$ to cn. During the $(cn+1)$th checkpoint interval, the MSS rejects the transmission of P_j's $HardCkpt^{cn}$ (see code IV, lines 38-46) so that the permanent checkpoint of P_j is not overwritten by the wrongly taken hard checkpoint. On P_j's part, if the transmission of its $HardCkpt^{cn}$ is rejected by the local MSS, P_j deletes $HardCkpt^{cn}$.

5.3.3. Untimely Delayed Acknowledgements

In our algorithm, P_i's dependency vectors $Recv_i$ and $LastRecv_i$ are maintained by the local MSS (say MSS_k). MSS_k maintains these variables by inspecting the piggybacked information in an ACK sent by P_i, but an untimely delayed ACK could be a problem during the checkpointing process. Take figure 4 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 the proposed algorithm we take the following policy (see code IV, lines 18–25): 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 figure 4(a) always happens. But if $ACK.m$ finally arrives and shows that figure 4(b) is true instead, the receipt of m is then included in $Recv_i$ of the cnth checkpoint interval.

5.4. Rollback recovery

When a failure occurs, all the processes should roll back to the latest recovery line. Assume the latest recovery line is numbered as cn. Each process first checks whether its $HardCkpt^{cn}$ is still in the local disk. If $HardCkpt^{cn}$ is found, the process can roll back to the state of $HardCkpt^{cn}$ because the content of $HardCkpt^{cn}$ is identical to $PermCkpt^{cn}$. Otherwise, the process requests its $PermCkpt^{cn}$ from the HA or FA. Note that a wrongly taken $HardCkpt^{cn}$ of a process, as described in section 5.3.2, will not be used for recovery because the process will delete the checkpoint as soon as the transmission of $HardCkpt^{cn}$ is rejected.

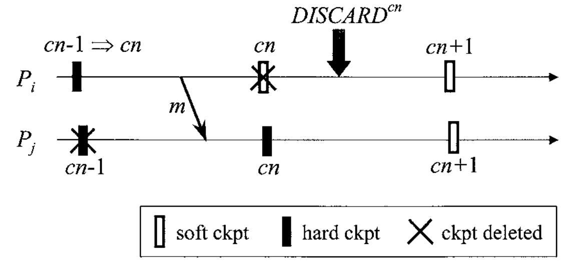


Figure 3. A possible scenario that the delivery of $DISCARD^{cn}$ for P_j is delayed.

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

5.5. Proof of correctness

Lemma 2. If a process P_i receives a message from another process P_j during the $(cn-1)$th checkpoint interval 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 the $(cn-1)$th checkpoint interval. Since P_i depends on P_j, the initiator also transitively depends on P_j during the $(cn-1)$th checkpoint interval. 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 checkpoints with checkpoint number 0 obviously form a globally consistent checkpoint. Assume there are N permanent checkpoints with checkpoint number k and they form a globally consistent checkpoint. In the proposed algorithm, if a process P_i receives a message m from another process P_j during the kth checkpoint interval, 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 $DISCARD^{k+1}$ during its $(k+1)$th checkpoint interval. From lemma 1 we know P_j's local MSS does not receive the ACK of $DISCARD^{k+1}$

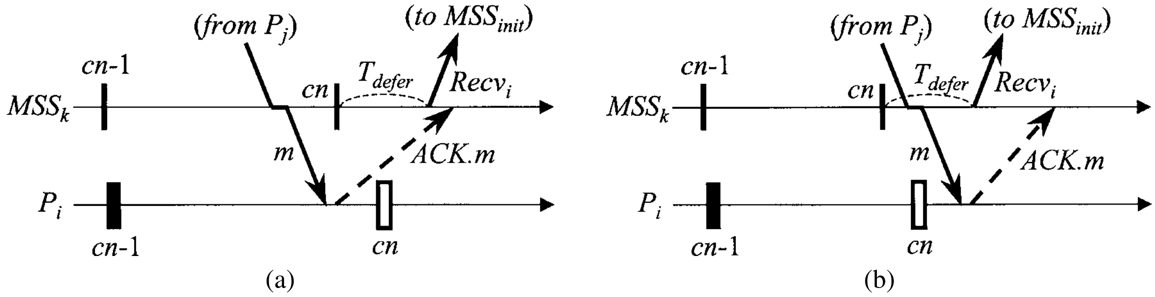


Figure 4. The ACK of m arrives later than MSS_k has sent $Recv_i$ to MSS_{init} (a) receipt of m is in $(cn-1)$th checkpoint interval of P_i; (b) receipt of m is in the cnth checkpoint interval of P_i.

at the end of its $(k + 1)$th checkpoint interval. As a result, $\mathit{RejectCP}_j$ is set to $k + 1$ and the preceding permanent checkpoint $\mathit{PermCkpt}^k$ of P_j is renumbered as $\mathit{PermCkpt}^{k+1}$.

(1.2) P_j receives $\mathit{DISCARD}^{k+1}$ during its $(k + 1)$th checkpoint interval. In this case, P_j discards its $\mathit{SoftCkpt}^{k+1}$ and the preceding permanent checkpoint $\mathit{PermCkpt}^k$ of P_j is renumbered as $\mathit{PermCkpt}^{k+1}$.

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

Case 2: If $P_j \notin \mathit{S\_Discard}^{k+1}$, P_j does not receive $\mathit{DISCARD}^{k+1}$ and turns its $\mathit{SoftCkpt}^{k+1}$ into $\mathit{HardCkpt}^{k+1}$ at the end of its $(k + 1)$th checkpoint interval. After that, P_j's $\mathit{HardCkpt}^{k+1}$ is saved as P_j's $\mathit{PermCkpt}^{k+1}$. From the proposed algorithm, P_j must send m before it takes $\mathit{SoftCkpt}^{k+1}$. Otherwise, P_i will take $\mathit{SoftCkpt}^{k+1}$ before processing m, which makes m been received within P_i's $(k + 1)$th checkpoint interval. As a result, no matter P_i's $\mathit{PermCkpt}^k$ is renumbered as $\mathit{PermCkpt}^{k+1}$ or P_i's $\mathit{HardCkpt}^{k+1}$ is saved as $\mathit{PermCkpt}^{k+1}$, there is no orphan message between P_i's $\mathit{PermCkpt}^{k+1}$ and P_j's $\mathit{PermCkpt}^{k+1}$.

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

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

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

5.6. Performance of our algorithm

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 [8].
- N_{dep}: the average number of processes on which a process depends $(1 \leqslant N_{dep} \leqslant N - 1)$.
- $C_{wireless}$: 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.

5.6.1. Blocking time

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

5.6.2. Number of new permanent checkpoints

In section 5.3.3, we described that if there is an unacknowledged message like the scenario depicted in figure 4, the *MSS* presumes the case in figure 4(a) always happens. That is, the receipt of message m from P_j is included in the *Recv* vector of P_i's $(cn - 1)$th checkpoint interval. If it turns out later that figure 4(b) is true instead, then there is a chance that P_j and P_j-dependent processes should not have been included in the final dependency with the initiator. The consequence is that there may be additional soft checkpoints be turned into hard ones, 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.

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

5.6.4. Comparison with other algorithms

Table 1 compares the performance of our algorithm with the algorithms in [3,8,13]. 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

Table 1
Performance comparison.

Algorithm	Blocking time	Checkpoints	Messages
Koo–Toueg [8]	$N_{min} \times T_{ckpt}$	N_{min}	$3 \times N_{min} \times N_{dep} \times (C_{wired} + C_{wireless})$
Neves–Fuchs [13]	$\sigma + 2\rho_{MH}T - td_{min}$	N	$2 \times N \times C_{wireless}$
Cao–Singhal [3]	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	$\approx N_{min}$	$\approx N \times C_{wired} + (N - N_{min}) \times (C_{wired} + C_{wireless})$

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 the wired network with high bandwidth.

6. Conclusions

In this paper we 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 a large mobile computing system 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.

References

[1] A. Acharya and B.R. Badrinath, Checkpointing distributed applications on mobile computers, in: *Proceedings of International Conference on Parallel and Distributed Information Systems* (September 1994) pp. 73–80.

[2] G. Cao and M. Singhal, On the impossibility of min-process non-blocking checkpointing and an efficient checkpointing algorithm for mobile computing systems, in: *Proceedings of the 27th International Conference on Parallel Processing* (August 1998) pp. 37–44.

[3] G. Cao and M. Singhal, Mutable checkpoints: A new checkpointing approach for mobile computing, IEEE Transactions on Parallel and Distributed Systems 12(2) (2001) 157–172.

[4] F. Cristian and F. Jahanian, A timestamp-based checkpointing protocol for long-lived distributed computations, in: *Proceedings of the IEEE International Symposium on Reliable Distributed Systems* (September 1991) pp. 12–20.

[5] E.N. Elnozahy, D.B. Johnson and W. Zwaenepoel, The performance of consistent checkpointing, in: *Proceedings of the 11th Symposium on Reliable Distributed Systems* (October 1992) pp. 39–47.

[6] E. Elnozahy, L. Alvisi, Y.M. Wang and D.B. Johnson, A survey of rollback recovery protocols in message passing systems, Technical report CMU-CS-99-148, School of Computer Science, Carnegie Mellon University (June 1999).

[7] H. Higaki and M. Takizawa, Checkpoint-recovery protocol for reliable mobile systems, in: *Proceedings of the IEEE Symposium on Reliable Distributed Systems* (October 1998) pp. 93–99.

[8] R. Koo and S. Toueg, Checkpointing and rollback-recovery for distributed systems, IEEE Transactions on Software Engineering (January 1987) 23–31.

[9] Y. Morita and H. Higaki, Hybrid checkpoint protocol for supporting mobile-to-mobile communication, in: *Proceedings of the IEEE International Conference on Information Networking* (January 2001) pp. 529–536.

[10] S. Neogy, A. Sinha and P.K. Das, Checkpoint processing in distributed systems software using synchronized clocks, in: *International Conference on Information Technology: Coding and Computing* (April 2001) pp. 555–559.

[11] N. Neves and W.K. Fuchs, Using time to improve the performance of coordinated checkpointing, in: *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (September 1996) pp. 282–291.

[12] N. Neves and W.K. Fuchs, Adaptive recovery for mobile environments, Communications of the ACM (January 1997) 68–74.

[13] N. Neves and W.K. Fuchs, Coordinated checkpointing without sirect coordination, in: *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (September 1998) pp. 23–31.

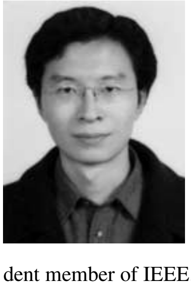
[14] T. Park and H.Y. Yeom, An asynchronous recovery scheme based on optimistic message logging for mobile computing systems, in: *Proceedings of the International Conference on Distributed Computing Systems* (April 2000) pp. 436–443.

[15] T. Park, N. Woo and H.Y. Yeom, An efficient recovery scheme for mobile computing environments, in: *IEEE International Conference on Parallel and Distributed Systems* (June 2001) pp. 53–60.

[16] R. Prakash and M. Singhal, Low-cost checkpointing and failure recovery in mobile computing systems, IEEE Transactions on Parallel and Distributed Systems 7(10) (1996) 1035–1048.

[17] K.F. Ssu, B. Yao, W.K. Fuchs and N. Neves, Adaptive checkpointing with storage management for mobile environments, IEEE Transactions on Reliability 48(4) (December 1999) 315–324.

[18] Z. Tong, R.Y. Kain and W.T. Tsai, A low overhead checkpointing and rollback recovery scheme for distributed systems, in: *Proceedings of the 8th Symposium on Reliable Distributed Systems* (October 1989) pp. 12–20.



Chi-Yi Lin received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1995. From August–December 2000, he was a visiting researcher in AT&T Labs-Research at New Jersey. He is currently working toward the Ph.D. degree at the Department of Electrical Engineering, National Taiwan University. His research interests include fault tolerant distributed/mobile computing systems, and the dissemination of information in wireless networks. He is a student member of IEEE.
E-mail: cylin@cc.ee.ntu.edu.tw

Szu-Chi Wang received the B.S. degree in computer science and information engineering from National Taiwan University in 1995. He is a Ph.D. candidate in the Department of Electrical Engineering at National Taiwan University. His research interests include distributed systems, fault-tolerant systems, and the dissemination of information in wireless networks.
E-mail: wsc@lion.ee.ntu.edu.tw

Sy-Yen Kuo received the B.S. (1979) in electrical engineering from National Taiwan University, the M.S. (1982) in electrical and computer engineering from the University of California at Santa Barbara, and the Ph.D. (1987) in computer science from the University of Illinois at Urbana-Champaign. Since 1991 he has been with National Taiwan University, where he is currently a professor and the Chairman of Department of Electrical Engineering. He spent his sabbatical year as a visiting researcher at AT&T Labs-Research, New Jersey from 1999 to 2000. He was the Chairman of the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan from 1995 to 1998, a faculty member in the Department of Electrical and Computer Engineering at the University of Arizona from 1988 to 1991, and an engineer at Fairchild Semiconductor and Silvar-Lisco, both in California, from 1982 to 1984. In 1989, he also worked as a summer faculty fellow at Jet Propulsion Laboratory of California Institute of Technology. His current research interests include mobile computing and networks, dependable distributed systems, software reliability, and optical WDM networks.

Professor Kuo is an IEEE Fellow. He has published more than 180 papers in journals and conferences. He received the distinguished research award (1997–2001) from the National Science Council, Taiwan. He was also a recipient of the Best Paper Award in the 1996 International Symposium on Software Reliability Engineering, the Best Paper Award in the simulation and test category at the 1986 IEEE/ACM Design Automation Conference (DAC), the National Science Foundation's Research Initiation Award in 1989, and the IEEE/ACM Design Automation Scholarship in 1990 and 1991.
E-mail: sykuo@cc.ee.ntu.edu.tw