JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 17, 549-573 (2001)

# Efficient Checkpoint-based Failure Recovery Techniques in Mobile Computing Systems*

CHENG-MIN LIN AND CHYI-REN DOW
*Department of Information Engineering*
*Feng Chia University*
*Taichung, Taiwan 407, R.O.C.*
*E-mail:{cmlin, crdow}@pluto.iecs.fcu.edu.tw*

Conventional distributed and domino effect-free failure recovery techniques are inappropriate for mobile computing systems because each mobile host is forced to take a new checkpoint (based on coordinated checkpointing). Otherwise, multiple local checkpoints may need to be stored in stable storage (based on communication-induced checkpointing). Hence, this investigation presents a novel domino effect-free failure recovery technique that combines the merits of the above two checkpointing technologies for mobile computing systems. The algorithm is a three-phase protocol that ensures a consistent checkpoint. The first phase uses a coordinated checkpointing protocol among mobile support stations. In the second phase, a communication-induced checkpointing protocol is used between each mobile support station and its mobile hosts. In the last phase, each mobile support station sends a checkpoint request to its mobile host which hasn't received any message from the mobile support station during the second phase. Numerical results are provided which compare the proposed algorithm with both a quasi-synchronous failure recovery algorithm and a hybrid checkpoint recovery algorithm for mobile computing systems. According to the comparison, our scheme outperforms other schemes in terms of checkpoint overhead. Moreover, the proposed algorithm has several merits: domino effect-free, nonblocking, twice the checkpoint size, and scalability.

***Keywords:*** mobile computing systems, checkpointing, failure recovery, domino effect, consistent global checkpoints

## 1. INTRODUCTION

Although a mobile computing system consists of static and mobile hosts, a distributed computing system consists of only static hosts. And while checkpointing techniques in distributed computing systems have received considerable attention [2], these techniques cannot be readily applied to mobile computing systems due to certain characteristics of mobile computing systems. The latter include mobility, low power consumption, disconnection, doze mode, low bandwidth, and limited memory.

In distributed systems, a critical problem with checkpointing/recovery is the domino effect [14], which may force the system to restart from the initial state due to cascading rollback propagation. Hence, distributed checkpointing/recovery algorithms without the domino effect must be designed. Distributed and domino effect-free checkpoint-

Received May 3, 1999; revised December 17, 1999 & March 20, 2000; accepted June 12, 2000.
Communicated by Yi-Bing Lin.
*This work is partially supported by National Science Council of the Republic of China under contract NSC-88-2213-E035-039.

ing/failure recovery techniques can be classified into two categories: coordinated and communication-induced checkpointing. Coordinated checkpointing ensures a consistent global checkpoint while it only needs to maintain twice the global checkpoint size. Except when the lowest checkpoint size is required, recovery and garbage collection are both simple. However, coordinated techniques force each mobile host to either take a new checkpoint or to block the underlying computation during checkpointing. Hence, it reduces process autonomy when taking checkpoints. For instance, these techniques send many additional messages to coordinate a consistent global checkpoint, forcing nodes to take a checkpoint. The coordination overhead is high due to mobility and energy capacity of mobile computing systems, and hence, coordinated techniques are inappropriate for mobile computing systems. Although the latter (communication-induced checkpointing) allows mobile hosts to take a checkpoint independently, each mobile host may needs to store multiple local checkpoints in a stable storage. These checkpoints are used to obtain a consistent global recovery line when a failure occurs. The result is that recovery and garbage collection are both complex. Therefore, communication-induced checkpointing techniques are inappropriate because of the limited memory in mobile computing systems.

In light of the above discussion, this work presents a novel domino effect-free failure recovery algorithm. A three-phase hybrid checkpointing protocol is proposed to ensure consistent checkpoints. In the first phase, moving the task of synchronization from mobile hosts (*MH*s) to mobile support stations (*MSS*s) allows for the reduction of complexity of synchronization overhead from O($n$) to O($m$), where $n$ and $m$ denote the number of *MH*s and *MSS*s, respectively. In the second phase, utilizing communication-induced techniques reduces synchronization overhead. A *MSS* does not immediately send a request message to its *MH*s. In this phase, the *MSS* sets a timer $T_{lazy}$ when it receives a request message from the coordinator. In the third phase, when $T_{lazy}$ has timed out, the *MSS* sends a checkpoint request to each *MH* which has not received any message since the *MSS* received the request message.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes a system model used for mobile computing systems. Section 4 introduces a novel checkpointing/recovery algorithm. Section 5 presents an analysis of the checkpoint method and Section 6 presents the numerical results. Finally, conclusions are made in Section 7.

## 2. RELATED WORK

Coordinated checkpointing techniques can be classified into two categories: blocking and nonblocking algorithms. The former forces all relevant processes to block their computation and focuses on minimizing the number of synchronization messages and the number of checkpoints during the checkpointing process. In the latter, processes do not need to be blocked during checkpointing because a checkpoint sequence number is used to identify orphan messages. Therefore, the performance of nonblocking algorithms is superior to that of blocking algorithms. In order to obtain a consistent global checkpoint, the coordinator initially sends a request message to other processes when its checkpoint interval is up. Next, each process blocks its computation and takes a tentative checkpoint when it has received the request message. Then, each process sends a response message

to the coordinator when it has finished the checkpoint. Finally, the coordinator sends a committed message to all processes after receiving the reply messages. Each process deletes its previous permanent checkpoint and changes the tentative checkpoint into a permanent checkpoint. As explained above, a consistent global checkpoint is ensured and only twice the global checkpoint size is required.

In the *Sync-and-Stop* (SNS) algorithm [12], the coordinator initially defines a consistent cut, and then stops all processes to take their local checkpoints. This algorithm is the simplest consistent checkpointing technique and is a blocking algorithm. However, when the number of *MH*s is very large, it is inappropriate for mobile systems because its synchronization overhead is O($n$), where $n$ denotes the number of *MH*s. The *Chandy-Lamport* (CL) [1] checkpointing technique solves the major problem of the SNS algorithm where all processors must be frozen during checkpointing. Although CL uses a nonblocking technique, the algorithm sends O($n^2$) extra messages if the network, consisting of $n$ hosts, is fully connected, making the algorithm unacceptable for large values of $n$. Most coordinated checkpointing techniques, such as SNS and CL, can be adopted for failure recovery. After failures, all nodes are rolled back to their last checkpoint. Prakash and Singhal's algorithm [13] combines both blocking and nonblocking techniques. Their algorithm forces only a minimum number of processes to take checkpoints, and does not block the underlying computation during checkpointing. In addition, the algorithm only requires a minimal set of nodes to be rolled back to the last checkpoint after a failure occurs.

Independent checkpointing techniques do not require system-wide coordination and therefore may scale better [2]. Although simple and having a low run-time overhead, they may incur a high recovery overhead due to the domino effect. To prevent the domino effect, the system preserves a consistent global snapshot. Xu and Netzer [11] introduced the notion of *zigzag paths*, a generalization of Lamport's happened-before relation [6]; they also presented a necessary and sufficient condition in order to obtain a consistent global snapshot. The same investigation also presented an adaptive independent checkpointing algorithm to reduce rollback propagation [25]. That algorithm forced the controller to take an additional local checkpoint, called adaptive checkpoint [25] or forced checkpoint [10], in order to prevent useless checkpoints.

Used to avert the domino effect in independent checkpointing, communication-induced checkpointing techniques can be classified into two categories: model-based and index-based checkpointing [2]. The former uses a communication pattern to determine whether or not to take a checkpoint. The latter uses a checkpoint index piggybacked on each message. The model-based checkpointing techniques include CASBR [4], CAS [7, 23-24], CBR [17], NRAS [17], FDI [20], and FDAS [20]. Although model-based checkpointing techniques ensure that no useless checkpoint occurs, it may cause many forced checkpoints. In index-based checkpointing, upon receiving a message with a piggybacked index greater than the local index, the receiver must take a forced checkpoint to avert an inconsistent state.

Manivannan and Singhal proposed a quasi-synchronous checkpointing algorithm [8-10], abbreviated as MS in this paper. This algorithm is simple and has a merit of asynchronous checkpointing, low overhead, and a merit of synchronous checkpointing, low recovery time. In the MS algorithm, each process takes checkpoints independently, called basic checkpoints. Checkpoints triggered by message receptions are called forced

checkpoints. The checkpoint index is increased by one after taking a basic or a forced checkpoint. When process $P_i$ receives a message $m$, with a piggybacked information $index_j$ from process $P_j$, and $P_i$'s $index_i$ is smaller than $index_j$, a forced checkpoint is taken to advance the recovery line. Although the *MS* algorithm has a low checkpoint overhead, it has to maintain multiple checkpoints. Wang and Fuchs [22] presented the notation of laziness to provide a compromise between checkpoint overhead and rollback distance. Increasing the laziness increases the rollback distance and, possibly, reduces the checkpoint overhead.

In 1998, Higaki and Takizawa [5] proposed a hybrid checkpointing protocol for mobile computing systems. The algorithm integrates the advantages of both coordinated and independent checkpointing for mobile computing systems. However, the algorithm has two defects. First, using independent checkpointing protocol may cause the domino effect. Second, coordinated and independent checkpointing protocols perform independently in mobile support stations and mobile hosts, and do not negotiate with each other. Therefore, it is difficult to obtain consistent global checkpoints.

When a failure occurs, independent and communication-induced checkpointing techniques must identify a consistent global checkpoint. Many graph models have been used in independent checkpointing to find a consistent global checkpoint for recovery, and to obsolete checkpoints for garbage collection (e.g., the antecedence graph [3], the roll-back-dependency graph (R-graph) [20], and the checkpoint graph [21]). Using these graph models, the minimum or maximum consistent checkpoint can be obtained. However, such techniques produce complex rollback recovery and garbage collection. Instead of using additional messages to coordinate other processes to take a consistent checkpoint, Tong *et al*. used synchronized checkpoint clocks to fulfill the coordination [18]. Moreover, no additional message is required to produce a consistent global state. By using such a technique, a process takes a checkpoint and waits for a period of time that equals the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process in the system. Although capable of saving the cost of sending messages for coordination, the protocol waits a longer time to obtain a consistent global checkpoint. In addition, such clock deviation techniques are inappropriate for mobile computing systems because a *MH* can be in the doze-mode or disconnection mode.

## 3. SYSTEM MODEL

A mobile computing system consists of a set of mobile hosts, $MH_i$, where $0 \le i \le n\text{-}1$ and $n$ is the number of mobile hosts in the system, that communicate with the *MSS* through wireless channels. There exists no shared memory or common clock among these nodes and the communication between the nodes is by message-passing only. We assume that wireless channels and logical channels are all FIFO order.

Fig. 1 illustrates an example of a mobile computing system. If a *MH* moves to the cell of another base station, a wireless channel to the old *MSS* is disconnected and a wireless channel in the new *MSS* is allocated. Such an action is called *handoff*. For instance, $MH_3$ moves from $MSS_3$ to $MSS_2$. $MH_3$ initially sends a *leave*($MH_3$) message to $MSS_3$ and, then, sends a *join*($MH_3$, $MSS_3$) message to $MSS_2$. Next, $MSS_2$ sends a *handoff*($MH_3$) message to $MSS_3$. Finally, $MSS_3$ forwards all information of $MH_3$ to $MSS_2$.

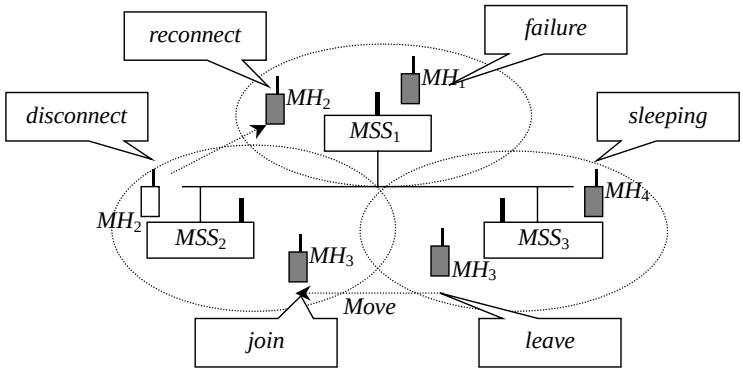


Fig. 1. An example of a mobile computing system.

A *MH* may voluntarily disconnect from mobile computing networks. The *MH* does not send and receive any message when it is in a disconnected state. For instance, $MH_2$ in Fig. 1 initially sends a *disconnect*($MH_2$) message to inform $MSS_2$ not to send any message to it. Then, if other mobile hosts send messages to $MH_2$, $MSS_2$ stores the message into its buffer. When $MH_2$ moves to $MSS_1$ and sends a *reconnect*($MH_2$, $MSS_2$) message to $MSS_1$, $MSS_1$ informs $MSS_2$ to transmit all $MH_2$'s messages in its buffer.

We can model the execution of a program in distributed systems with a computer program, P = <*E*, →>, where *E* is a finite set of *events* and → is the *happened before* relation defined over *E*. First, events are the execution of the following instructions, such as send, receive, or checkpoint operations. We assume that each node of the system periodically takes a checkpoint in fixed time. The *initial* and *i*th checkpoints in process $P_a$ are denoted by $C_{a,0}$ and $C_{a,i}$, respectively. The time between two consecutive checkpoints, $C_{a,i}$ and $C_{a,i+1}$, is called *checkpoint interval* $I_{a,i}$ (including $C_{a,i}$ but not $C_{a,i+1}$).

The happened-before relation, denoted by '→', shows the precedence of two events. For example, A→B shows that event A happens before event B. The happened-before relation exists from $C_{a,i}$ to $C_{b,j}$, descibed by $C_{a,i} \rightarrow C_{b,j}$, if and only if:

1. $a = b$ and $j = i + 1$,
2. $a \neq b$ and $C_{a,i}$ is the *i*th checkpoint before sending message *m* by process $P_a$, and $C_{b,j}$ is the *j*th checkpoint after receiving message *m* by process $P_b$, and
3. there is a checkpoint $C_{c,k}$ such that $C_{a,i} \rightarrow C_{c,k}$ and $C_{c,k} \rightarrow C_{b,j}$.

Fig. 2 depicts a time-space diagram of the system in Fig. 1 from the perspectives of *MSS* and *MH*. To reduce energy consumption, a *MH* may voluntarily perform a sleeping instruction and transform its status into a doze mode in which the *MH* cannot send any message to other mobile hosts; however, it can receive messages from others. The *MH* can be waked after receiving a message. According to Fig. 2, $MH_4$ operates in a "doze mode," and later it is waked by message $m_6$. We assume that the system is fail-stop [16]. For instance, when a failure occurs in $MH_1$, the system stops and restarts the process of all *MH*s after the recovery.

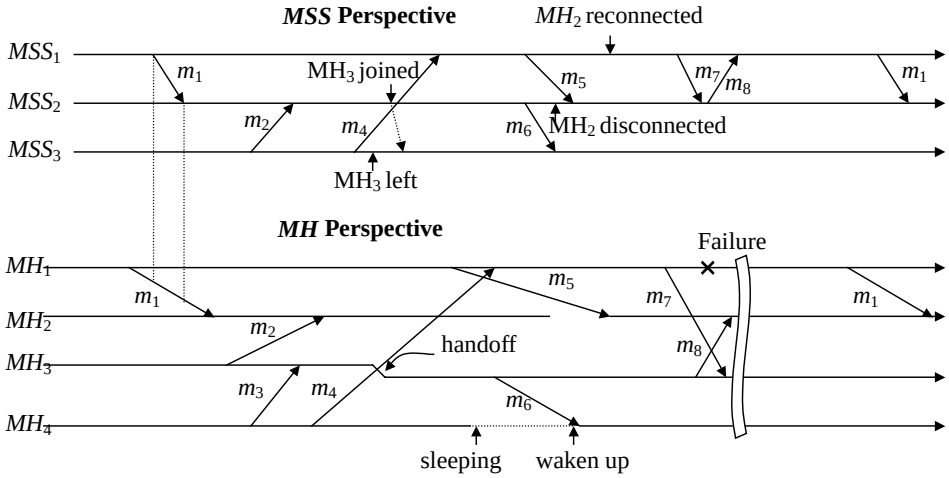


Fig. 2. Time-space diagram of the mobile computing system in Fig. 1.

## 4. A RECOVERY ALGORITHM

In this section, a novel domino effect-free failure recovery algorithm is presented. The algoirthm is derived from a two-tier architecture of cellular mobile systems. We initially address our underlying notions, and then present the algorithm. Finally, the validity of our algorithm is verified.

### 4.1 Underlying Notions

Our hybrid checkpointing technique combines the coordinated checkpointing technique with the communication-induced checkpointing technique. Coordinated checkpointing is used among *MSS*s, while communication-induced checkpointing is used between a *MSS* and its *MH*s. The coordinating actions only run among *MSS*s, making our algorithm a nonblocking one. In addition, the synchronization overhead is quite low because the overhead is reduced from O($n$) to O($m$), where $n$ and $m$ denote the number of *MH*s and *MSS*s, respectively. Using communication-induced checkpointing between *MSS*s and *MH*s allows us to achieve quasi-synchronization [8-10], and effecively reduces synchronization overhead because additional messages are reduced to an acceptable range. Although our algorithm also takes forced checkpoints, the induction ratio R, where R is a ratio of the total number of checkpoints to the number of basic checkpoints [8], is close to one. Therefore, the checkpoint overhead of our algorithm is low because R nearly equals one. In addition, the complexity of synchronization overhead is reduced to O($m$).

Table 1 compares the results of our hybrid algorithm with coordinated and communication-induced checkpointing techniques. Only communication-induced checkpointing techniques do not require any additional message. In contrast, coordinated checkpointing techniques largely use additional messages to implement the checkpointing action, and so require many additional messages. The hybrid checkpointing technique uses the tech-

**Table 1. Comparison of three checkpointing strategies.**

| Strategy | Additional messages | Maintained checkpoints | Disturbed *MH*s | Power consumption | Mobility |
|---|---|---|---|---|---|
| Coordinated checkpointing | Many | Two | Many | High | Poor or Fair |
| Communication-induced checkpointing | No | Many | No | Low | Good |
| Our checkpointing | Few | Two | Few | Low | Good |

nique of lazy coordination, for instance $T_{lazy}$, to reduce additional messages. This requires only a few additional messages. In the checkpoint space, coordinated and hybrid checkpointing techniques belong to a two-phase commitment protocol that requires only twice the checkpoint space. However, the communication-induced checkpointing techniques have difficulty in determining a consistent global cut on the fly because it may require many checkpoints of each process after a failure occurs. A *MH* is disturbed if it receives an additional message causing it to wake up from a sleeping mode. Hence, communication-induced techniques cannot disturb any *MH*. Coordinated checkpointing techniques can disturb *MH*s because they create too many additional messages. Coordinated checkpointing techniques send many additional messages and may disturb *MH*s causing greater power consumption. Hence, the power consumed by coordinated checkpointing techniques is higher than that of the other two. From a mobility perspective, a coordinated action will fail if a coordinated *MH* is sleeping or disconnected since most coordinated checkpointing algorithms require all *MH*s to take checkpoints together. *MH*s send a request message to every *MH*, but some disconnected *MH*s cannot respond to the request until they are reconnected to this system.

Fig. 3 illustrates a hybrid checkpointing example. (1) Initially, the coordinator $MSS_1$ broadcasts a request message with a checkpoint index to $MSS_2$ and $MSS_3$. Next, each *MSS* sets up a timer $T_{lazy}$. (2) $MH_2$ ($MH_3$) takes a checkpoint before message $m_2$ ($m_3$) because it receives the message through $MSS_2$ ($MSS_3$) during $T_{lazy}$. In contrast, $MH_1$ ($MH_4$) takes a checkpoint when $T_{lazy}$ has timed out and it receives a checkpointing request message from $MSS_1$ ($MSS_3$). (3) Then, $MSS_2$ and $MSS_3$ send a response message to the coordinator $MSS_1$. $MSS_1$ sends committed messages to all *MSS*s after receiving all response messages from other *MSS*s. (4) When $MH_3$ moves from the $MSS_3$ cell to the $MSS_2$ cell, it performs a handoff procedure. (5) Although $MH_4$ is in the sleep mode, $m_6$ wakes it up to take a checkpoint. (6) $MH_2$ has to take a checkpoint before a disconnection operation is executed. When $MH_2$ is in the disconnection mode, it does not take a checkpoint despite receiving a computational message $m_5$ or a request message from $MSS_1$. (7) When a failure occurs in $MH_1$, $MH_1$ stops executing and sends a recovery message to $MSS_1$. (8) After $MSS_1$ receives the message, it sends a recovery message to all *MSS*s, and each *MSS* sends a recovery message to all *MH*s in its cell, causing the states of these four *MH*s to be rolled back to a consistent global cut, $\{C_{1,2}, C_{2,2}, C_{3,2}, C_{4,2}\}$. (9) After recovery, all *MH*s replay messages from their Log files, such as $m_5$, $m_6$, $m_7$ and $m_8$. (10) $MH_1$ rolls back to $C_{1,2}$ and sends $m_7$ again. However, $MH_3$ drops $m_7$ because it is a duplicate message.

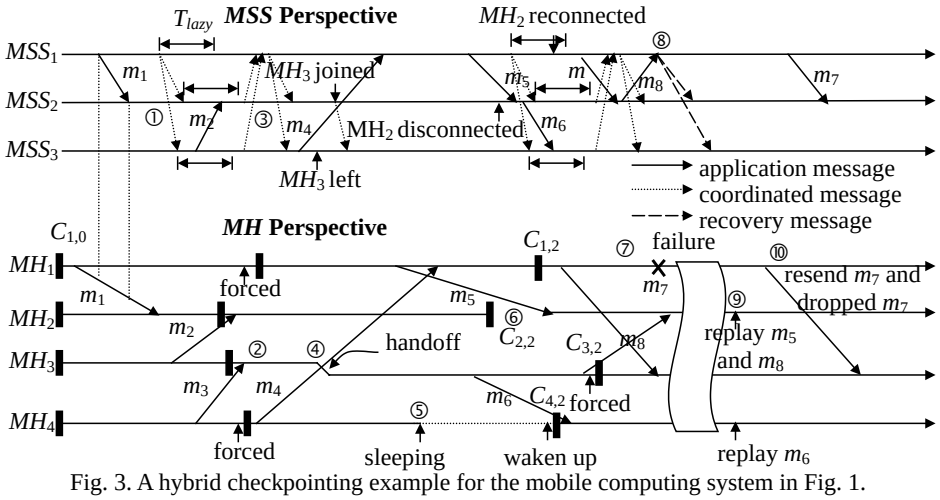


Fig. 3. A hybrid checkpointing example for the mobile computing system in Fig. 1.

### 4.2 Algorithm

Fig. 4 depicts the proposed failure recovery algorithm for mobile hosts. Details of the proposed algorithm are as follows. (1) Three integer variables, $index_i$, $inc_i$, and $rec\_line_i$, represent a checkpoint index, a current incarnation number, and a recovery line number in $MH_i$, respectively. (2) When the procedure checkpoint_here is called, $MH_i$ copies $CurrentCkpt_i$ to $BackupCkpt_i$, which are two checkpoint files. Then, a new checkpoint $CurrentCkpt_i$ is taken. Finally, the checkpoint index of *CurrentCkpt* is set to $index_i$. (3) When the procedure roll_back is called, $index_i$ is set to $rec\_line_i$, and checkpoint_here is called if $rec\_line_i$, is greater than or equal to $index_i$. Next, the status of $MH_i$ is rolled back to $BackupCkpt_i$ if the index of $BackupCkpt_i$ is greater than or equal to $rec\_line_i$. Otherwise, the *MH* is rolled back to $CurrentCkpt_i$. (4) When recovering after failures, $MH_i$ initialy rolls back $CurrentCkpt_i$ status, and then restores $index_i$ to $CurrentCkpt_i$, increases the incarnation number [8], sets recovery line number to $index_i$, and sends a recovery message to $MSS_p$. (5) When receiving a computaional message *M*, $MH_i$ takes a new checkpoint before *M* and sets local checkpoint index $index_i$ to *M.index* if $index_i$ is less than *M.index*. Note that no useless checkpoint occurs with communication-induced checkpointing techniques. In addition, procedure roll_back is called if the incarnation number attached in message *M* is larger than $inc_i$. Finally, $MH_i$ processes the message *M*. (6) If $MH_i$ receives a request message from $MSS_p$ and *M.index* is greater than $index_i$, then procedure checkpoint_here(*M.index*) is called. In addition, procedure roll_back is called if the incarnation number attached in message *M* is larger than $inc_i$. (7) When $MH_i$ receives a recovery message and the incarnation number attached in message *M* is larger than $inc_i$, then procedure roll_back is called. (8) When $MH_i$ sends a computational message to $MSS_p$, the local checkpoint index $index_i$, the incarnation number $inc_i$, and the recovery line number $rec\_line_i$ can be attached to the computational message. With this approach, a consistent cut is ensured regardless of when a handoff occurs. For example, $MH_i$ initially moves to $MSS_p$ and then sends a *computational* message *M* with $index_i$ to another $MH_j$. The local checkpoint $index_j$ is set to *M.index* if the received

*M.index* is greater than $index_j$. Thus, a new checkpoint index is propagated to *MH*s when a *MSS* forwards messages to the *MH*s in its cell. Notice that Fig. 4 does not use message logging [8].

**(1) Data Structure at $MH_i$**
$index_i$, $inc_i$, $rec\_line_i$ : integer (: = 0)
**(2) Procedure checkpoint_here**
$BackupCkpt_i := CurrentCkpt_i$;
Take checkpoint $CurrentCkpt_i$;
$CurrentCkpt_i.index = index_i$;
**(3) Procedure roll_back**
**If** $rec\_line_i > index_i$ **then**
$index_i := rec\_line_i$; checkpoint_here;
**ElseIf** $BackupCkpt_i.index \geq rec\_line_i$ **then**
$index_i = BackupCkpt_i.index$; **Restore** $BackupCkpt_i$;
**ElseIf** $CurrentCkpt_i.index \geq rec\_line_i$ **then**
$index_i = CurrentCkpt_i.index$; **Restore** $CurrentCkpt_i$;
**End If**;
**(4) Recovery initiated by $MH_i$ after failure**
**Restore** $CurrentCkpt_i$;
$index_i := CurrentCkpt_i.index$;
$inc_i := inc_i + 1$;
$rec\_line_i := index_i$;
**send** $recovery(inc_i, rec\_line_i)$ to $MSS_p$;
**(5) When $MH_i$ receives a *computational* message *M* from $MH_j$**
**If** $M.index > index_i$ **then**
$index_i = M.index$; checkpoint_here;
**End If**;
**If** $M.inc > inc_i$ **then**
$inc_i := M.inc$; $rec\_line_i := M.rec\_line$; roll_back;
**End If**;
Process message *M*;
**(6) When $MH_i$ receives a *request* message *M* from $MSS_p$**
**If** $M.index > index_i$ **then**
$index_i = M.index$; checkpoint_here;
**End If**;
**If** $M.inc > inc_i$ **then**
$inc_i := M.inc$; $rec\_line_i := M.rec\_line$; roll_back;
**End If**;
**(7) When $MH_i$ receives a *recovery* message *M* from $MSS_p$**
**If** $M.inc > inc_i$ **then**
$inc_i := M.inc$; $rec\_line_i := M.rec\_line$; roll_back;
**End If**;
**(8) When $MH_i$ sends a message *M* to $MH_j$**
$M.(index, inc, rec\_line) := (index_i, inc_i, rec\_line_i)$;
Send message *M* to $MSS_p$;

Fig. 4. The failure recovery algorithm for MHs.

We now give some examples of the failure algorithm for *MH*s. Assume in a mobile computing system that there are four *MH*s, $MH_0$, $MH_1$, $MH_2$ and $MH_3$. $MH_0$ is located in the cell of $MSS_0$, and its $(index, inc, rec\_line)_0$ variables are $(3, 1, 3)_0$. Six cases can be

discussed as follows. (1) $MH_0$ receives a computational message $M$ with (3, 1, 3) from $MH_1$. No additional checkpoint is taken because $index_0$ equals $M.index$ and $inc_0$ equals $M.inc$. (2) When $MH_0$ receives a computational message $M$ with (4, 1, 3) from $MH_2$, $MH_0$ takes a checkpoint because $index_0$ is less than $M.index$. (3) When $MH_0$ receives a computational message $M$ with (3, 2, 3) from $MH_3$, $MH_0$ rolls back to $CurrentCkpt_0$ because $inc_0$ is less than $M.inc$. (4) When $MH_0$ receives a request message $M$ with (4, 1, 3) from $MSS_0$, $MH_0$ takes a checkpoint because $index_0$ is less than $M.index$. (5) When $MH_0$ receives a recovery message $M$ with (3, 2, 3) from $MSS_0$, $MH_0$ rolls back to $CurrentCkpt_0$. (6) When $MH_0$ receives a recovery message $M$ with (3, 0, 3) from $MSS_0$, $MH_0$ ignores this message because $inc_0$ is larger than $M.inc$.

Fig. 5 depicts the proposed failure recovery algorithm based on hybrid checkpointing techniques for *MSS*s. Details of the algorithm are described as follows. (1) $MSS_p$ must maintain several variables, including a local checkpoint index, a variable storing the number of committed messages, a current incarnation number, and a recovery line number, a member queue, a request queue, a disconnected queue, and a disconnected buffer, denoted by $index_p$, $CommittedCnt_p$, $inc_p$, $rec\_line_p$, $MQueue_p$, $RQueue_p$, $DQueue_p$, and $DBuffer_p$, respectively. The initial values of $index_p$, $CommittedCnt_p$, $inc_p$, and $rec\_line_p$ are set to zero. The initial values of other variables are set to empty. (2) With periodical checkpointing, if $MSS_p$ is the coordinator, it initially increases $index_p$ and sends a request message with the index to all *MSS*s except itself after a fixed time interval. (3) When $MSS_p$ receives a *request* message $M$ from $MSS_q$, $MSS_p$ performs the following steps if it discovers $index_p$ is less than $M.index$. $MSS_p$ initially sets $index_p$ to $M.index$, then sets a timer $T_{lazy}$ that is used to delay the coordination action. Finally, $MSS_p$ copies $MQueue_p$ to $RQueue_p$, where $MQueue_p$ represents all *MH*s in its cell and $RQueue_p$ is used to store the *MH*s that still do not take a new checkpoint after receiving the request message from $MSS_p$. (4) When $MSS_p$ receives a *recovery* message $M$ from $MSS_q$, $MSS_p$ performs the following steps if it discovers $inc_p$ is less than $M.index$. $MSS_p$ sets $inc_p$ to $M.inc$. Then it sets $index_p$ and $rec\_line_p$ to $M.rec\_line$. Finally, $MSS_p$ sends a *recovery* message $M$ to all *MH*s in its cell. (5) When $MSS_p$ receives a *recovery* message $M$ from $MH_j$, $MSS_p$ performs the same procedure as in step 4 except it sends a recovery message to all *MSS*s instead of *MH*s. (6) When $T_{lazy}$ times out, $MSS_p$ sends a *request* message to every *MH* whose identification is still in $RQueue_p$. On the other hand, if *MH*s do not receive any messages from $MSS_p$, no checkpoint is taken by the communication-induced checkpointing. Finally, $MSS_p$ sends a response message to the coordinated *MSS*. (7) When receiving a *computational* message $M$ being send from $MH_i$ to $MH_j$, $MSS_p$ performs the following steps. Initially, $index_p$, $inc_p$ and $rec\_line_p$ are updated to the corresponding values of $M$ if $index_i$ is less than $M.index$. If a greater index is reserved, no useless checkpoint is taken. Next, $MSS_p$ deletes $MH_j$'s identification from $RQueue_p$ if it is still in $RQueue_p$. Moreover, $M$ is put into $DBuffer_p$ if $MH_j$ disconnects from the computing system. Finally, $MSS_p$ forwards $M$ to $MH_j$ when it discovers $MH_j$ is in its cell; otherwise, $MSS_p$ searches for $MH_j$ in the $MSS_q$ which forwards $M$ to the $MSS_q$. (8) When receiving a *response* message, $MSS_p$ initially increases variable $CommittedCnt_p$, then checks the variable to see whether or not it equals the total number of *MSS*s. If they are equal, $MSS_p$ broadcasts a *committed* message to each *MSS* and resets the variable to zero. (9) When receiving a *committed* message from coordinator $MSS_q$, $MSS_p$ sets $rec\_line_p$ to $index_p$, i.e., a new recovery line can be obtained.

**(1) Data Structure at $MSS_p$**
$index_p$, $CommittedCnt_p$, $rec\_line_p$, $inc_p$ : integer (: = 0)
$MQueue_p$, $RQueue_p$, $DQueue_p$, $DBuffer_p$: Queue (: = ϕ)
**(2) When it is time for $MSS_p$ to take a basic checkpoint**
$index_p := index_p + 1$;
$M.index = index_i$;
send a *request* message $M$ to all *MSSs*;
**(3) When $MSS_p$ receives a *request* message $M$ from $MSS_q$**
**If** $M.index > index_p$ **then**
$index_p := M.index$;
Set a lazy coordinated timer $T_{lazy}$;
$RQueue_p = MQueue_p$;
**end if**;
**(4) When $MSS_p$ receives a *recovery* message $M$ from $MSS_q$**
**If** $M.inc > inc_p$ **then**
$inc_p := M.inc$;
$index_p := rec\_line_p := M.rec\_line$;
send a *recovery* message $M$ to all *MHs* in its cell;
**end if**;
**(5) When $MSS_p$ receives a *recovery* message $M$ from $MH_j$**
**If** $M.inc > inc_p$ **then**
$inc_p := M.inc$;
$index_p := rec\_line_i := M.rec\_line$;
send a *recovery* message $M$ to all *MSSs*;
**End If**;
**(6) When $T_{lazy}$ has timed out for $MSS_p$**
Send a *request* message to $MH_i$, $\forall i \in RQueue_p$;
Send a *response* message to coordinated *MSS*;
**(7) When $MSS_p$ receives a *computational* message $M$ being send from $MH_i$ to $MH_j$**
**If** $M.index > index_p$ **then**
$index_p := max(M.index, index_p)$;
$inc_p := max(M.inc, inc_p)$;
$rec\_line_p := max(M.rec\_line, rec\_line_p)$;
**End If**
**If** $j \in RQueue_p$ **then** delete it from $RQueue_p$;
**If** $j \in DQueue_p$ **then** put the message $M$ into $DBuffer_p$;
**If** $MH_j$ is in the $MSS_p$ cell **then** deliver message $M$ to $MH_j$;
**Else** search $MH_j$ and deliver message $M$ to $MSS_q$ that $MH_j$ is in the $MSS_q$ cell;
**(8) When $MSS_p$ receives a *response* message $M$ from $MSS_q$**
$CommittedCnt_p = CommittedCnt_p + 1$;
**If** $CommittedCnt_p$ = MAX_MSSs **then** broadcast a *committed* message to all *MSSs and reset C* $CommittedCnt_p$; $rec\_line_p = index_p$;
**(9) When $MSS_p$ receives a *committed* message $M$ from $MSS_q$**
$rec\_line_p = index_p$;

Fig. 5. The failure recovery algorithm for MSSs.

Examples of the failure recovery algorithm for MSSs are now presented. Assume that there are three *MSS*s ($MSS_0$, $MSS_1$, and $MSS_2$) in a mobile computing system. Their variables $(index, inc, rec\_line)_i$ are $(3, 1, 3)_i$, and assume that $MSS_0$ is the coordinator. Three cases are discussed as follows. (1) When time is up, $MSS_0$ initially sets $index_0$ to 4 and sends a request message with $index_0$ to $MSS_1$ and $MSS_2$. After receiving the request message, $MSS_1$ and $MSS_2$ update their index to 4. Also, the three *MSS*s set a lazy coordinated timer $T_{lasy}$. When the time is up, $MSS_1$ and $MSS_2$ send a local committed message to

$MSS_0$. After receiving both local committed messages, $MSS_0$ sends a global committed message to $MSS_1$ and $MSS_2$. Finally, these *MSS*s update their $rec\_line_i$ to 4. (2) When $MSS_1$ receives a recovery message $M$ with (3, 2, 2) from a *MH*, $MSS_1$ updates (3, 1, 3) to (3, 2, 2) and sends a recovery message to $MSS_0$ and $MSS_2$ because $inc_1$ is less than $M.inc$. After receiving the request message, $MSS_0$ and $MSS_2$ update their variable, after which all *MSS*s send a request message to their *MH*s. (3) When $MSS_2$ receives a computational message $M$ with (4, 1, 3), $MSS_2$ updates its index to 4.

### 4.3 Proof of Correctness

**Lemma 1:** If $MH_i$ and $MH_j$ take checkpoints $C_i$ and $C_j$, respectively, and $index_i = index_j$, then the two checkpoints are consistent.

***Proof:*** Assume that $MH_i$ and $MH_j$ take checkpoints $C_i$ and $C_j$, respectively, and $index_i = index_j$, but the two checkpoints are inconsistent. We learn from [11] that there is a zigzag path *ZP* between $C_i$ and $C_j$. Hence, several cases are presented below.

**Case 1:** If the length of *ZP* is one, $MH_i$ sends a message $M$ with $index_i$ to $MH_j$ after $C_i$ is taken. $MH_j$ receives $M$ before $C_j$ is taken. Thus, the $index_i$ attached in $M$ should be equal to $index_j - 1$. This case cannot occur because $MH_j$ takes $C_j$ before receiving the message if $index_i$ equals $index_j$.

**Case 2:** If *ZP* has length two, there are three possible situations. First, after $C_i$, $MH_i$ sends a message $M_1$ to $MH_k$. $MH_k$ initially takes a checkpoint $C_k$ before receiving $M_1$, and then before $C_j$, $MH_k$ sends a message $M_2$ to $MH_j$. The situation is the same as in Case 1, so it cannot occur. Second, $MH_k$ does not take a checkpoint $C_k$ before receiving $M_1$, but this situation has the same result as the first situation, so it cannot occur. Finally, the order between $M_1$ and $M_2$ in $MH_k$ is reversed. No inconsistent state can occur if $index_i$ is equal to or less than $index_k$. $MH_k$ will take a checkpoint $C_k$ before receiving $M_1$ if $index_i$ is greater than $index_j$. Thus, *ZP* is broken. In summary, Case 2 cannot occur.

**Case 3:** If *ZP* has length $n$, these checkpoints are consistent. In the previous analysis when ZP is two, we have proven that these checkpoints cannot be inconsistent. We can reduce $n$ to $n - 2$ repeatedly until $n$ is one or two, and if $MH_i$ and $MH_j$ take checkpoints $C_i$ and $C_j$, respectively, and $index_i = index_j$, then the two checkpoints are consistent. ❑

**Lemma 2:** If $MH_i$ and $MH_j$ take checkpoints $C_i$ and $C_j$, respectively, and $index_i = index_j$, checkpoints $C_i$ and $C_j$ are taken by the same checkpoint initiation.

***Proof:*** Increasing a checkpoint index can only occur in a coordinator *MSS*. According to this result and Lemma 1, $C_i$ and $C_j$ are consistent so we can prove that the two checkpoints belong to the same checkpoint initiation and the coordinator *MSS* starts the initiation. ❑

**Lemma 3:** A global checkpoint can be obtained within a finite amount of time after the coordinator starts checkpointing.

***Proof:*** The algorithm must perform the following steps to take a checkpoint:

1. A coordinator *MSS* sends a request message to other *MSS*s.
2. When $MSS_i$ receives a request message, it sets timer $T_{lazy}$.
3. When $T_{lazy}$ has timed out, $MSS_i$ sends a request message to the *MH*s in its cell which did not receive any message during $T_{lazy}$.
4. After $MSS_i$ receives all checkpoints coming from the *MH*s of its cell, it sends a response message to the coordinator *MSS*.
5. After receiving all response messages, the coordinator *MSS* sends a committed message to all *MSS*s.

Each step can be achieved in a finite amount of time. Hence, we prove that a global checkpoint can be obtained in a finite amount of time. ❑

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

***Proof:*** According to Lemma 1, two checkpoints with same index are consistent. Furthermore, according to Lemma 1 and Lemma 2 we infer that a consistent global checkpoint consists of local checkpoints with the same index. We also know from Lemma 3 that a consistent global checkpoint can be obtained in a finite amount of time. However, according to the three Lemmas, the algorithm only ensures a consistent global checkpoint if *MH*s do not move or disconnect. Hence, we need to consider the following cases:

**Case 1:** For all *MH*s about to disconnect, the algorithm takes a checkpoint before the disconnection. Hence, its checkpoint is used as an initial checkpoint after a reconnection. Therefore, no inconsistent state occurs due to the disconnection.

**Case 2:** For all *MH*s in doze mode, the algorithm sends a request message to wake up them to take a checkpoint. Hence, all *MH*s in doze mode can not cause an inconsistent state.

**Case 3:** When a *MH* moves from a cell of $MSS_i$ to another cell of $MSS_j$, $MSS_j$ sends a hand-off message to $MSS_i$ to obtain the *MH*'s checkpoints. Hence, a consistent state is kept when a *MH* moves.

As discussion above, we prove that the algorithm ensures a consistent global checkpoint even if it is a special case. ❑

**Theorem 2.** The algorithm rolls back all necessary processes to a global consistent cut when a failure occurs.

***Proof:*** An orphaned message can cause an inconsistent state after a failure recovery. An orphan message *M* occurs when the states of $MH_i$ and $MH_j$ are rolled back to checkpoints $C_{i,a}$ and $C_{j,b}$, where *a* and *b* are checkpoint indices of $MH_i$ and $MH_j$, and *M* is sent after $C_{i,a}$, but it is received before $C_{j,b}$. There are two cases depending on whether the message is sent before or after the failure. If it occurs before the failure, $MH_i$ sends message *M* to

$MH_j$ after $C_{i,a}$, and $MH_j$ receives $M$, and then takes $C_{j,b}$. Based on step 5 in Fig. 4, we prove that $a$ is greater than $b$. If it occurs after the failure and recovery, based on the procedure roll_back, $b$ must be equal to or less than $a$. On the other hand, our failure recovery algorithm asserts that the states of $MH_i$ and $MH_j$ are rolled back to $C_{i,a}$ and $C_{j,b-1}$. Hence, the situation above cannot occur. In addition, according to the procedure roll_back in Fig. 4, unnecessary processes will not roll back when *M.rec_line* is greater than $index_j$. ❑

**Theorem 3**. The algorithm is domino effect-free.

***Proof:*** According to Theorems 1 and 2, every global checkpoint is consistent. Furthermore, we infer that no useless checkpoints are taken so no propagation rollback occurs after failures, and therefore, the algorithm belongs to a domino effect-free algorithm. ❑

## 5. ANALYSIS OF CHECKPOINT OVERHEAD

In this section we analyze the checkpoint overhead of our recovery algorithm. The execution time of the program without checkpointing is denoted as $T_E$; $T_c$ denotes the execution time of the program with checkpointing. The percentage of checkpoint overhead is

$$OH = \frac{T_c - T_E}{T_E} \times 100. \quad (1)$$

The execution time of a *MH* is divided into several intervals. Let $\Gamma$ denote the expected execution time of an interval; a checkpoint interval and a checkpoint time are denoted as $I_c$ and $t_c$, respectively. The overhead includes cost of saving a checkpoint to non-volatile storage and uploading the checkpoint to the *MSS*. The execution time is $\Gamma = I_c$ if no failure occurs during the interval. This gives $I_c - t_c$ units of useful computation time during each interval. If one or more failures occur within an interval, then the execution time is larger than $I_c$. So, we modify (1) as follows.

$$OH = (\frac{\Gamma}{I_c - t_c} - 1) \times 100. \quad (2)$$

Fig. 6 illustrates a 4-state discrete Markov chain based on the 3-state discrete Markov chain [19]. The Markov chain shows how the expected execution time $\Gamma$ of a single interval can be evaluated in a *MH*. State 0 is the initial state when an interval being executing. For our algorithm, there are two recovery cases. In the first case, a transition from state 0 to state 1 denotes that a *MH* rolls back to the last checkpoint state $C_{last}$ when a failure occurs within the inteval (i.e. its rollback distance is one). In the second case, a transition from state 0 to state 2 rolls a *MH* back to a previous checkpoint $C_{last-1}$ when a failure occurs(i.e. its rollback distance is two). If a failure occurs in a *MH* after entering state 1, the *MH* rolls back only to the last checkpoint $C_{last}$, which corresponds to a transition from state 1 back to itself. Otherwise, a *MH* rolls back to the state prior to the last checkpoint $C_{last-1}$, corresponding to a transition from state 1 to state 2. Finally, when state 3, called a sink state, is entered, the interval is complete.

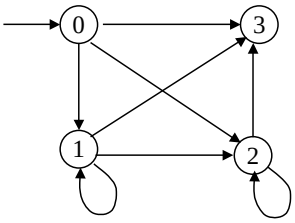

Fig. 6. Markov chain.

Each transition, from state X to state Y in the Markov chain (X, Y) has an associated transition probability $P_{XY}$ and a cost function $K_{XY}$. $K_{XY}$ is the expected time spent in state X before entering state Y. State 0 is the initial state when an interval begins executing while state 3 is the terminal state. If no failure occurs in the interval, transition (0, 3) is made. The probability of transition (0, 3) is

$$P_{03} = e^{-\lambda_f I_c} \tag{3}$$

and the cost is

$$K_{03} = I_c. \tag{4}$$

If a failure occurs during the interval, then transition (0, 1) is made. The probability of this is

$$P_{01} = a(1 - P_{03}), \text{ where } 0 < a \le 1 \tag{5}$$

where $a$ is the ratio of the number of *MH*s whose rollback distance is one to the total number of *MH*s. The cost of this transition, from the beginning of the interval to point of failure, is the expected duration. Therefore,

$$K_{01} = \int_0^{I_c} \frac{\lambda_f t e^{-\lambda_f t}}{1 - e^{-\lambda_f I_c}} dt = \frac{1}{\lambda_f} - \frac{I_c e^{-\lambda_f I_c}}{1 - e^{-\lambda_f I_c}}. \tag{6}$$

Furthermore, if a MH rolls back to the previous state, caused by other *MH*s, the probability of transition (0, 2) is

$$P_{02} = b(1 - P_{03}), \text{ where } 0 \le b \le 1, \tag{7}$$

where $b$ is the ratio of the number of *MH*s whose rollback distance is two, to the total number of *MH*s. The relation between $a$ and $b$ is

$$a + b = 1. \tag{8}$$

The cost of this transition consists of two parts. Part one, $K_{01}$, represents the cost for a *MH* to roll back to the last checkpoint. The second part comes from the *MH* having to

roll back to the previous checkpoint if the rollback distance is greater than one. Hence, the cost is

$$K_{02} = K_{01} + (I_c - t_c). \tag{9}$$

While in state 1, if no failure occurs before the next checkpoint is established, then a (1, 3) transition is made. The execution time also consists of two parts: recovery time $t_r$ and a useful computation time $I_c$ - $t_c$, resulting in a transition probability

$$P_{13} = e^{-\lambda_f (t_r + I_c - t_c)} \tag{10}$$

and cost

$$K_{13} = t_r + I_c - t_c \tag{11}$$

While in state 2, if no failure occurs before the next checkpoint is established, then a (2, 3) transition is made. The execution time consists of two parts: recovery time $t_r$ and two times the useful computation, i.e., $2(I_c - t_c)$. The probability of the transition is

$$P_{23} = e^{-\lambda_f (t_r + 2(I_c - t_c))} \tag{12}$$

and the cost is

$$K_{23} = t_r + 2(I_c - t_c) \tag{13}$$

However, if another failure occurs while in state 1, and the last checkpoint is rolled back, then a (1, 1) transition is made. The probability of this transition is

$$P_{11} = 1 - P_{13} - P_{12}. \tag{14}$$

The cost of this transition, which is evaluated similarly to $K_{01}$, is

$$K_{11} = \int_0^{t_r + I_c - t_c} \frac{\lambda_f t e^{-\lambda_f t}}{1 - e^{-\lambda_f (t_r + I_c - t_c)}} dt = \frac{1}{\lambda_f} - \frac{(t_r + I_c - t_c) e^{-\lambda_f (t_r + I_c - t_c)}}{1 - e^{-\lambda_f (t_r + I_c - t_c)}} . \tag{15}$$

Therefore, if another failure occurs after entering state 1 and the checkpoint $C_{last\text{-}1}$ is rolled back, then a transition (1, 2) is made. For this transition the probability is

$$P_{12} = P_{02} \tag{16}$$

and the cost is

$$K_{12} = K_{02} . \tag{17}$$

However, if another failure occurs after entering state 2 and the checkpoint $C_{last\text{-}1}$ is rolled back, then a transition is made from state 2 back to itself. The probability of this transition is

$$P_{22} = 1 - P_{23} - P_{21} \tag{18}$$

and the cost is

$$K_{22} = \int_0^{t_r+2(I_c-t_c)} \frac{\lambda_f t e^{-\lambda_f t}}{1-e^{-\lambda_f (t_r+2(I_c-t_c))}} dt = \frac{1}{\lambda_f} - \frac{(t_r+2(I_c-t_c))e^{-\lambda_f (t_r+2(I_c-t_c))}}{1-e^{-\lambda_f (t_r+2(I_c-t_c))}}. \tag{19}$$

We assume that no two consecutive failures exist in the same interval. The expected time Γ can be shown to be

$$\Gamma = P_{03}K_{03} + P_{01}(K_{01} + K_{13}) + P_{02}(K_{02} + K_{23}). \tag{20}$$

Substituting the expressions for various cost values and transition probabilities into the above expression for Γ yields the following:

$$\begin{aligned}\Gamma &= I_c e^{-\lambda_f I_c} + a(1-e^{-\lambda_f I_c})\left(\frac{1}{\lambda_f} - \frac{I_c e^{-\lambda_f I_c}}{1-e^{-\lambda_f I_c}} + t_r + I_c - t_c\right) + \\ &\quad b(1-e^{-\lambda_f I_c})\left(\frac{1}{\lambda_f} - \frac{I_c e^{-\lambda_f I_c}}{1-e^{-\lambda_f I_c}} + I_c - t_c + t_r + 2(I_c - t_c)\right) \\ &= \left(1-e^{-\lambda_f I_c}\right)\left(\frac{1}{\lambda_f} + a(I_c - t_c) + 3b(I_c - t_c) + t_r\right).\end{aligned} \tag{21}$$

Substituting (2) into (21) we obtain:

$$OH = \left(\frac{(1-e^{-\lambda_f I_c})(\frac{1}{\lambda_f} + a(I_c - t_c) + 3b(I_c - t_c) + t_r)}{I_c - t_c} - 1\right) \times 100\%. \tag{22}$$

If we assume that a rollback distance of one and a rollback distance of two have equal probability (i.e. $a = b = 0.5$), we have

$$OH = \left(\frac{(1-e^{-\lambda_f I_c})(\lambda_f^{-1} + 2(I_c - t_c) + t_r)}{I_c - t_c} - 1\right) \times 100\% \ . \tag{23}$$

## 6. EXPERIMENTAL RESULTS

This section summarizes the experimental results. A stochastic simulation and numerical results are included.

### 6.1 Simulation Model

We used C++ to develop a stochastic simulator to simulate a mobile computing system that consists of *MSS*s and *MH*s. Three checkpointing algorithms, MS, HT and LD, are evaluated in this experiment. MS is a communication-induced algorithm, HT is a hybrid checkpoint algorithm described in Section 2, and LD represents our algorithm introduced in Section 4. For MS checkpointing [8], a "next" index interval, denoted $I_N$, must be specified. The interval is set to $I_c$ to obtain the same checkpoint interval used in coordinated schemes. The interval is used to increase index. For LD checkpointing, time $T_{lazy}$ is required and is included in checkpoint interval $I_c$. Each *MH* sends a message to other MH according to $\lambda_M$, where $\lambda_M$ is the parameter of an exponential distribution. The checkpoint arrival rate for each *MH* has a poisson distribution with parameter $\lambda_c$. The mean arrival rate is $1/I_c$. The rate includes basic checkpoints but not forced checkpoints; it also has an exponential distribution. For the whole system, the failure rate is Poisson distributed with parameter $\lambda_f$. When a failure occurs, the system randomly selects a MH as a faulty node and stops the whole system to perform a recovery. If $\lambda_f$ is zero, there is no failure in the simulation system. Table 2 presents the default value of the simulation parameters used in next subsection.

**Table 2. Simulator Parameters.**

| Parameters | Description | Default Value |
|---|---|---|
| $N$ | *The number of* MHs | 256 |
| $T_E$ | The execution time without checkpointing | 1000 |
| $T_{lazy}$ | Lazy time for LD checkpointing | 20 |
| $I_N$ | Next interval for MS checkpointing | 100 |
| $I_c$ | Checkpoint interval | 100 |
| $t_c$ | Saving and uploading time for a checkpoint | 10 |
| $t_R$ | Residence time | 100 |
| $t_d$ | Disconnect time | 10 |
| $t_s$ | Sleep time | 10 |
| $\lambda_M$ | Message sending rate with exponential distribution for each MH | 0.1 |
| $\lambda_c$ | Checkpoint arrival rate for each MH with Poisson process | 0.01 |
| $\lambda_f$ | Failure rate for the system with Poisson process | 0.0001 or 0 |
| $\lambda_d$ | Disconnecting rate with Poisson process | 0 |
| $\lambda_s$ | Sleeping rate with Poisson process | 0 |

### 6.2 Numerical Results

This section presents the numerical results from evaluating the checkpoint overhead according to the above simulation model. The evaluation is based on five parameters: computational time, checkpoint time, message sending interval, the number of *MH*s, and failure rates. Three algorithms, MS, HT, and LD are evaluated in these experiments. Each approach is evaluated both with and without failures. In the case of failures, the failure rate is set to 0.0001, except in Section 6.2.5.

### 6.2.1 Checkpoint overhead vs. computation time

Fig. 7 illustrates the average checkpoint overhead as a function of computation time. The average checkpoint overhead for the LD algorithm, the MS algorithm and the HT algorithm, without failure, are 10.7%, 10.6%, and 11.4%, respectively. The dotted line in this figure is the result of the analysis in Section 5. The overhead of our algorithm (LD) without failures is close to that of HT algorithm. The overhead of the MS algorithm is better than the others because it does not send any additional message for coordinating checkpoints. However, the overheads of the three algorithms without failures are about the same.

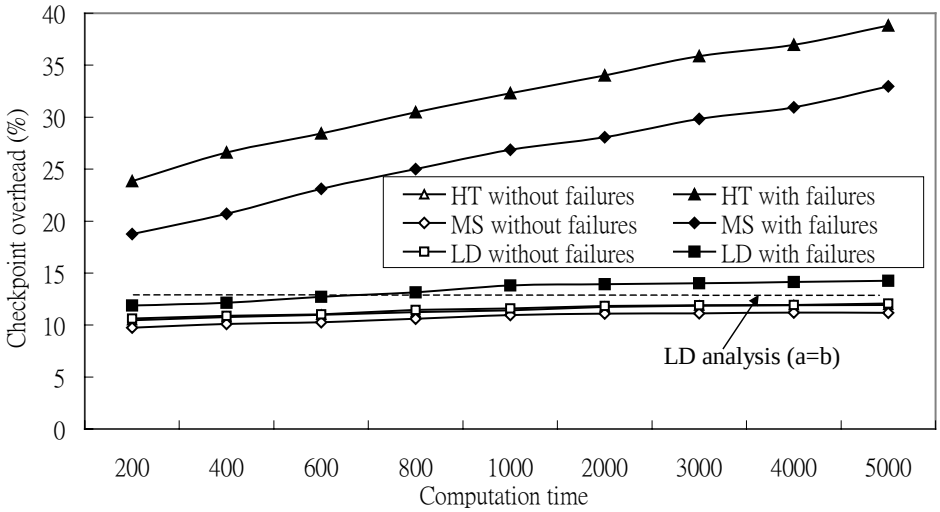


Fig. 7. Checkpoint overhead vs. computation time.

The average checkpoint overhead for the LD algorithm, the MS algorithm and the HT algorithm, with failures, the overhead is 13.3%, 26.2%, and 31.9%, respectively. The value of the checkpoint overhead is 12.7% based on equation 23. Furthermore, LD's checkpoint overhead is better than that of the MS and HT algorithms, especially when the computation time is high (i.e., a long running program). For instance, when the computation time is 5000 units, the overheads of the three algorithms, LD, MS, and HT, with failures are 14.2%, 33% and 38.8%, respectively. A good checkpointing algorithm is critical for long running programs, which makes LD an appropriate algorithm for long running programs in mobile systems. The MS algorithm has a higher checkpoint overhead than the LD algorithm because multiple rollbacks may occur. But HT algorithm has the highest checkpoint overhead because HT uses a synchronous method (i.e., all nodes must restart from the local checkpoints), while MS and LD use an asynchronous method. HT uses an asynchronous checkpointing scheme in *MH*s and a coordinated checkpointing scheme in *MSS*s. These two schemes cannot be smoothly used to coordinate a consistent global checkpoint.

### 6.2.2 Checkpoint overhead vs. checkpoint time

A service-oriented application generally requires a shorter checkpoint time than a computation application. Fig. 8 illustrates the overheads needed when checkpoint time increases from 1 to 10 units. Regardless whether there are failures or not, the checkpoint overhead increases when checkpoint time increases. The overheads of three algorithms without failures are nearly identical. However, when failures are present, LD is superior to both MS and HT. When the checkpoint time is ten, MS and HT with failures require 30.6% and 37% checkpoint overhead, respectively. However, only 13.8% overhead is required for LD with failures. Hence, the LD algorithm is more appropriate for programs in cases of failure.

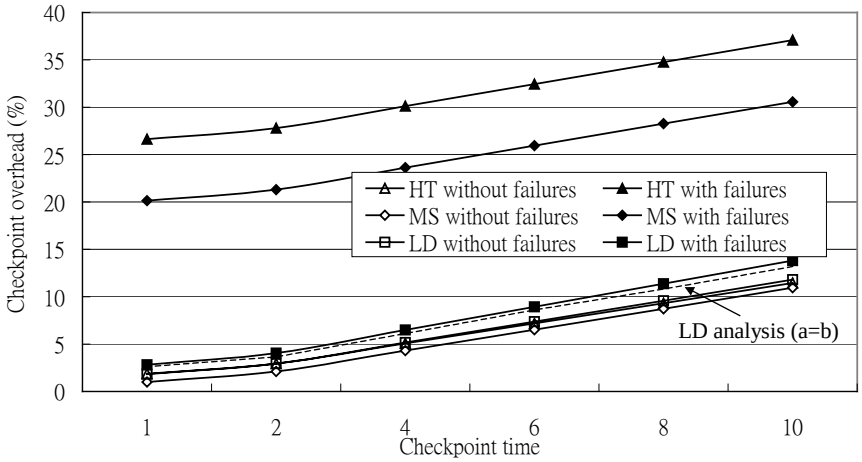


Fig. 8. Checkpoint overhead vs. checkpoint time.

### 6.2.3 Checkpoint overhead vs. message sending interval

Fig. 9 illustrates the relationship between checkpoint overhead and message sending interval. Generally speaking, programs with many messages have shorter sending intervals. Our algorithm generally outperforms MS and HT from a message interval point of view. For cases of failure in Fig. 9, LD is superior to both MS and HT. In failure free cases, the MS algorithm is superior to the HT and LD algorithms. Note that the checkpoint overheads of the LD and HT algorithms are insensitive to the sending interval. However, the checkpoint overhead for MS decreases slightly with increasing sending interval since the number of forced checkpoints decreases.

### 6.2.4 Checkpoint overhead vs. number of *MH*s

A better distributed checkpointing algorithm should be scalable. For example, checkpoint overhead can be limited to a reasonable range as the number of nodes increases. Fig. 10 illustrates the checkpoint overhead versus the number of *MH*s. The checkpoint overheads of LD, MS, and HT increase by 0.2%, 2%, and 3.2%, respectively,

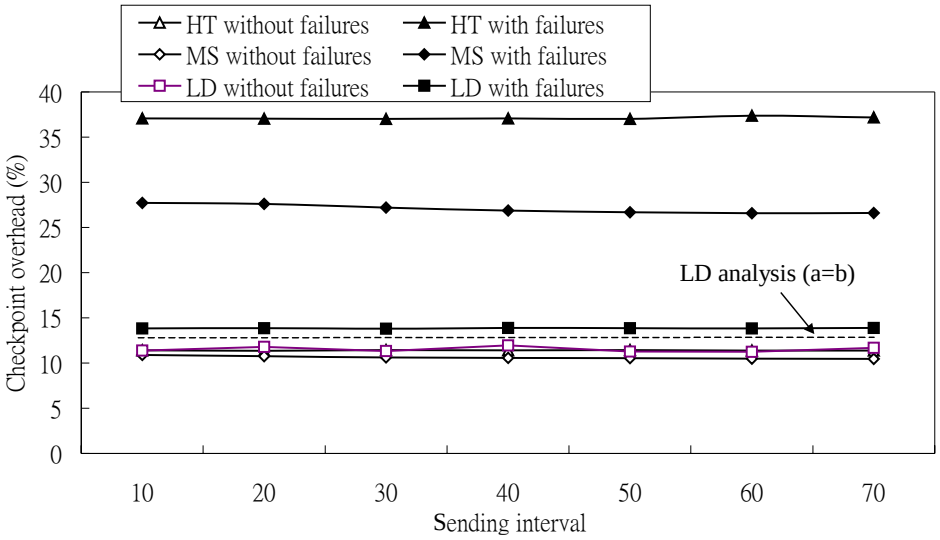


Fig. 9. Checkpoint overhead vs.sending interval.

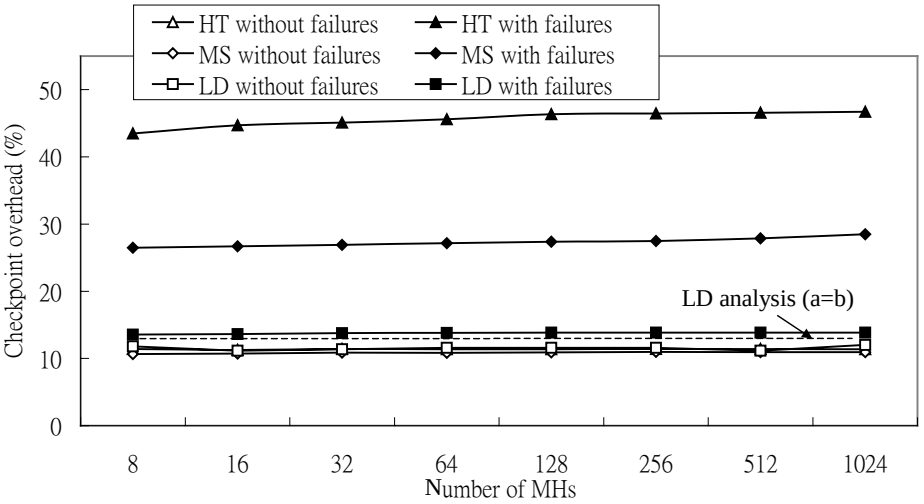


Fig. 10. Checnkpoint overhead vs. the number of MHs.

as the number of *MH*s increases from 8 to 1024. Hence, our algorithm is scalable. Moreover, Fig. 10 shows that the LD algorithm is better than both the MS and HT algorithms, especially when the number of *MH*s is large for cases of failure. According to this figure, for MS (HT) without and with failures needs an average overhead of 10.9% and 27.3% (11.4% and 45.6%); the LD algorithm needs only 11.5% and 13.8%.

### 6.2.5 Checkpoint overhead vs. failure rates

In this subsection, we observe how failure rates impact checkpoint overhead. Fig. 11 illustrates how higher failure rates dominant the checkpoint overhead. The checkpoint overheads of the HT and MS algorithms are sensitive to the failure rate due to multiple rollback propagation. In contrast, the increasing of checkpoint overhead for the LD algorithm increases by only 3.8% as the failure rate increases five folds. As a result, our algorithm is more suited to unreliable environments such as mobile computing systems.

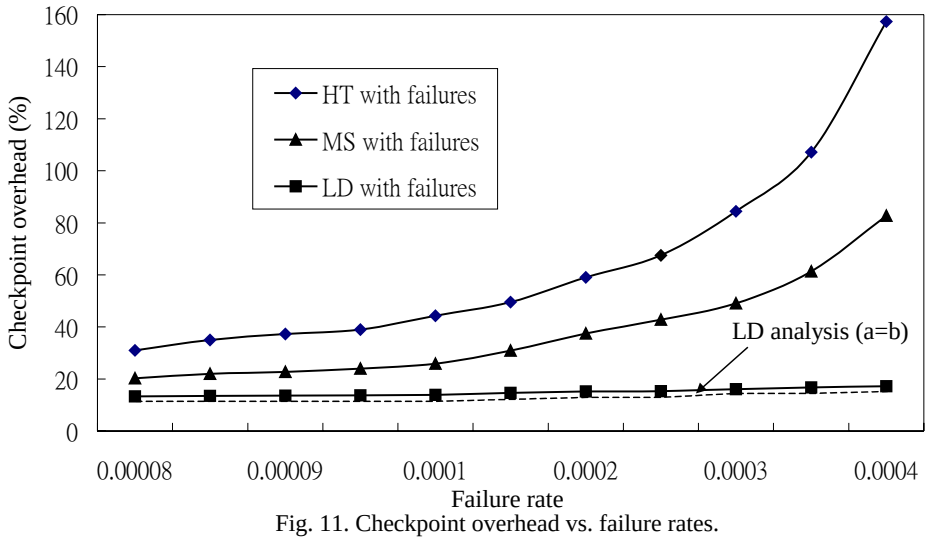


Fig. 11. Checkpoint overhead vs. failure rates.

## 7. CONCLUSIONS

The advantages of coordinated checkpointing algorithms are that they ensure a consistent global checkpoint and that they need only two checkpoints. However, these algorithms send many additional messages for coordination, thereby causing higher checkpoint overhead as the number of *MH*s increases. Furthermore, such algorithms can disturb many sleeping *MH*s and consume too much power for coordination. In contrast, communication-induced checkpointing algorithms do not disturb sleeping *MHs* and no additional messages are sent for coordination. Hence, the checkpoint overhead is sensitive to increasing the numbers of *MH*s. However, after failures many checkpoints must be taken to find a consistent global cut. As a result, the algorithms must maintain many checkpoints, and garbage collection is more difficult to implement in communication-induced checkpointing.

A stochastic simulation model is developed to compare our algorithm with two existing algorithms: MS and HT. For cases of failure, numerical results demonstrate that our algorithm outperforms others in terms of checkpoint overhead. Although the HT algorithm has a low checkpoint overhead for failure free cases, it cannot ensure that it is domino effect-free. In addition, it has a high checkpoint overhead for cases of failure.

Although MS is a communication-induced checkpointing algorithm, it must store multiple checkpoints for failure recovery. MS also has the lowest checkpoint overhead for failure-free cases because no additional message is sent. In contrast, our algorithm uses a three-phase coordination for checkpointing, making it is easy to obtain the consistent global cut. The failure recovery algorithm in this investigation integrates the advantages of coordinated and communication-induced checkpointing algorithms. In summing, our algorithm has the following advantages.

1. It is a domino effect-free algorithm.
2. It has the smallest checkpoint overhead of all three algorithms in cases of failure.
3. Its checkpoint overhead is similar to HT's in failure-free cases.
4. Consistent global checkpoints can be ensured.
5. No mobile host will be blocked during checkpointing.
6. It requires only twice the checkpoint size.
7. It is scalable since the checkpoint overhead is insensitive to the number of *MH*s.
8. The power consumption is low.
9. It handles the disconnection problem of a mobile host.
10. Very few hosts in doze mode will be disturbed.

## ACKNOWLEDGMENT

The authors like to thank the National Science Council of the Republic of China for financially supporting this research under Contact No. NSC-88-2213-E035-039.

## REFERENCES

1. K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, 1985, pp. 3-75.
2. E. N. Elnozahy, D. B. Johnson, and Y. M. Wang, "A survey of rollback protocols in message-passing systems," Technical Report, CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.
3. E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, Vol. 41, No. 5, 1992, pp. 526-531.
4. J. Fowler and W. Zwaenepoel, "Causal distributed breakpoints," in *Proceeding of IEEE International Conference on Distributed Computing Systems*, 1990, pp. 134-141.
5. H. Higaki and M. Takizawa, "Checkpoint-recovery protocol for reliable mobile systems," in *Proceedings of 7th IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 93-99.
6. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications ACM*, Vol. 21, No. 7, 1978, pp. 558-565.
7. C. M. Lin and C. R. Dow, "Adaptive distributed breakpoint detection in mes-

sage-passing programs," in *Proceeding of the 4th Workshop on Compiler Techniques for High-Performance Computing* (*CTHPC*'98), 1998, pp. 151-157.
8. D. Manivannan and M. Singhal, "A low-overhead recovery technique using quasi-synchronous checkpointing," in *Proceedings of* 16*th International Conference on Distributed Computing Systems*, 1996, pp. 100-107.
9. D. Manivannan and M. Singhal, "Failure recovery based on quasi-synchronous checkpointing in mobile computing systems," Technical Report, OSU-CISRC-7/96-TR36, Dept. of Computer and Information Science, Ohio State University, July 1996.
10. D. Manivannan and M. Singhal," Quasi-synchronous checkpointing: models, characterization, and classification," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 7, 1999, pp. 703-713.
11. R. H. B. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 2, 1995, pp. 165-169.
12. J. S. Plank, "Effect checkpointing on MIMD architectures," PhD thesis, Department of Computer Science, Princeton University, 1993.
13. R. Prakash and M. Singhal, "Low-cost checkpointing and failure recovery in mobile computing systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 10, 1996, pp. 1035-1048.
14. B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, 1975, pp. 220-232.
15. D. L. Russell, "State restoration in systems of communicating processes," *IEEE Transactions on Software Engineering*, Vol. 6, No. 2, 1980, pp. 183-194.
16. R. D. Schlichting and F. B. "Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems*, Vol. 1, No. 3, 1983, pp. 222-238.
17. R. E. Strom, D. F. Bacon, and S. A. Yemini, "Volatile logging in n-fault-tolerant distributed systems," in *Proceedings of IEEE Fault-Tolerant Computing Symposium*, 1998, pp. 44-49.
18. Z. Tong , R. Y. Kain, and W. T. Tasi, "Rollback recovery in distributed systems using loosely synchronized clocks," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 2 1992, pp. 246-251.
19. Nitin H. Vaidya, "On checkpoint latency," Technical Report, 95-015, Dept. of Computer Science, Texas A&M University, March 1995.
20. Y. M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Transactions on Computers*, Vol. 46, No.4, 1997, pp. 456-468.
21. Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs, "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 5, 1995, pp. 546-554.
22. Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Proceedings of IEEE Symposium on Reliable Distributed Systems*, 1993, pp. 78-85.
23. K. L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Transactions on Computers*, Vol. 39, No. 4, 1990, pp. 460-469.
24. K. L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multi-

processors using private caches," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, 1990, pp. 231-240.
25. J. Xu and R. H. B. Netzer, "Adaptive independent checkpointing for reducing rollback propagation," in *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993, pp. 754-761.

**Cheng-Min Lin (林正敏)** was born in 1964. He received the B.S. and M.S. degrees in electronic engineering from Nation Taiwan Institute of Technology, Taipei, Taiwan, in 1989 and 1991, respectively. He is currently a graduate student for the Ph.D. degree in the Department of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan. His research interests include distributed computing, fault tolerance, mobile computing, and debugging tools.

**Chyi-Ren Dow (竇其仁)** was born in 1962. He received the B.S. and M.S. degrees in computer science and information engineering from National Chiao Tung University, Taiwan, in 1984 and 1988, respectively, and the M.S. and Ph.D. degrees in computer science from the University of Pittsburgh, USA, in 1992 and 1994, respectively. Currently, he is an Associate Professor in the Department of Information Engineering and Computer Science, Feng Chia Unversity. His research interests include mobile computing, distributed systems, and multimedia software engineering.