Chapter 14

DEPENDABLE DISTRIBUTED AND MOBILE COMPUTING – UTILIZING TIME TO ENHANCE RECOVERY FROM FAILURES

W. Kent Fuchs
fuchs@purdue.edu
Electrical and Computing Engineering, Purdue University, West Lafayette IN 47907

Nuno Neves
nuno@di.fc.ul.pt
Computer Science Department, University of Lisbon, Portugal

Kuo-Feng Ssu
ssu@ecn.purdue.edu
Coordinated Science Laboratory, University of Illinois, Urbana IL 61801

Abstract Mobile computing allows ubiquitous and continuous access to computing resources while users travel or work at a client's site. The flexibility introduced by mobile computing brings new challenges to dependability and fault tolerance. Failures that were rare with fixed hosts become common, and host disconnections make fault detection and message coordination difficult. This chapter describes checkpointing and failure recovery procedures that are well adapted to both distributed and mobile environments. The protocols use time to indirectly coordinate the creation of new global states and thereby avoid message exchanges. The mobile protocol uses two different types of checkpoints to adapt to network characteristics. Procedures for integrating adaptive mobile checkpointing with storage management are also described.

Keywords: checkpointing, mobile computing, dependable computing, fault tolerance, rollback recovery

1. INTRODUCTION

One effective way to recover from failures in distributed systems is to use checkpointing and rollback recovery. Typically, a checkpoint protocol periodically saves the state of the application in stable storage. After a failure, the application rolls back to the last state that was saved and starts its re-execution. Checkpoint protocols are usually divided into two groups, uncoordinated (Borg et al., 1989) (Johnson and Zwaenepoel, 1987) (Neves et al., 1994) (Strom and Yemini, 1985) (Wang and Fuchs, 1993) (Wang and Fuchs, 1992) (Yao et al., 1999) (Ssu et al., 1999) and coordinated (Chandy and Lamport, 1985) (Cristian and Jahanian, 1991) (Kim and Park, 1993) (Koo and Toueg, 1987) (Neves and Fuchs, 1996) (Neves and Fuchs, 1998a) (Plank, 1993). In uncoordinated checkpoint protocols, each process determines independently from the others the instant when its state should be saved. To avoid the domino effect, the checkpoint protocol also saves information about the messages that were exchanged among processes.

In coordinated checkpoint protocols, processes coordinate among themselves to determine which process states should be included in the application checkpoint. The coordination is necessary to guarantee that the application checkpoint is consistent and recoverable (section 2.2 explains these concepts). These protocols usually select one of the processes, the coordinator, to initiate the creation of the checkpoints and to ensure that each process saves its state (Elnozahy et al., 1992) (Plank, 1993) (Silva and Silva, 1992). This task is accomplished with the exchange of a set of messages. The protocol adds information to each message to detect in-transit messages. Whenever an in-transit message arrives, the protocol saves the message in stable storage, together with the state of the processes.

Both types of protocols have their own advantages. However, coordinated protocols have shown better performance than uncoordinated protocols when used with parallel applications (Elnozahy and Zwaenepoel, 1994) (Neves and Fuchs, 1998b). Additionally, coordinated protocols do not need any piece-wise determinism assumption about the execution of the processes (Goldberg et al., 1990) and can typically tolerate failures that affect multiple processes simultaneously. Nevertheless, previous coordinated protocols have performance and implementation costs that can be avoided. In a typical coordinated protocol, the coordinator has to exchange three messages with each process. This overhead can become significant if the number of processes increases and the network is slow. The addition of information to messages to detect in-transit messages can also be an important overhead, depending on the level at which the protocol is implemented. If the protocol is implemented in a library, the overhead can be considerable, because each message may have to be copied. The third overhead is related to the in-transit messages. A process has to make an access to stable storage to save each in-transit message that it receives.

Time-based protocols relying on synchronized clocks have been developed in the past to avoid the first overhead (Tong et al., 1989) (Cristian and Jahanian, 1991) (Ramanathan and Shin, 1993), the exchange of coordination messages. These protocols assume that processors' clocks are approximately synchronized, and use time to indirectly coordinate the checkpoint creation. Each process saves its state whenever the local clock signals a checkpoint.

This chapter presents a time-based protocol for distributed systems that also uses time to coordinate the checkpoint creation. However, this protocol does not rely on synchronized clocks. It uses a simple initiation procedure to set checkpoint timers for the different processes, and then saves new checkpoints whenever the timers expire. Contrary to the previous time-based protocols, it also avoids the two other overheads that were mentioned previously.

We demonstrate that time-based checkpointing protocols can also be utilized for mobile computing. Checkpoint protocols proposed in the past are not adequate for mobile environments due to disconnections. These protocols have either to exchange messages during the creation of an application checkpoint (Chandy and Lamport, 1985) (Plank, 1993), or during recovery to collect stored information (Elnozahy and Zwaenepoel, 1992) (Johnson and Zwaenepoel, 1990). Protocols previously proposed for mobile computing have relied on the foreign agent for storing the checkpoint (Acharya and Badrinath, 1994) (Pradhan et al., 1996). Previous protocols also have not adapted their behavior to the characteristics of the current network connection.

The time-based mobile checkpointing procedure of this chapter is designed to take into consideration the special characteristics of mobile environments. The protocol is able to store consistent recoverable states of the application without having to exchange messages. Processes use a local timer to determine the instants when new checkpoints are to be saved. The protocol uses two different types of process checkpoints to adapt to the current characteristics of the network and to provide differentiated recoveries.

Finally, storage management for foreign agents is examined. The integration of leasing and adaptive checkpointing is shown to enhance checkpointing performance through hierarchical storage management. The approach enables simple garbage collection, efficient management of limited storage resources, and the reduction of consecutive missed checkpoints.

2. RECOVERY IN DISTRIBUTED SYSTEMS

2.1 SYSTEM ENVIRONMENT

The distributed systems under consideration are composed of a set of nodes interconnected by a network. A node contains a processing unit, a local memory and a local hardware clock. Clocks do *not* need to be synchronized among nodes. However, it is assumed that local clocks drift from real time with a maximum drift rate, ρ. This assumption implies that local clocks have at most an error of $\rho(e-s)$ seconds at the end of the real-time interval [s,e] seconds. The bounded drift rate condition applied to the local clock of a node n is

$$(1-\rho)(e-s) \leq C_n(e) - C_n(s) \leq (1+\rho)(e-s)$$

where $C_n(t)$ is time returned by the local clock when the real-time is t. The bounded drift equation can be used to derive a maximum deviation between the expiration of two timers. If two timers are started in two nodes exactly at the same time with the same initial value T, then they will expire by at most $2\rho T/(1-\rho^2)$ seconds from each other (we will approximate this value by $2\rho T$). Drift rates are in the order of 10^{-5} or 10^{-6} for most quartz clocks that are available in commodity computers, and for high precision clocks ρ is in the order of 10^{-7} or 10^{-8} (Cristian and Fetzer, 1994).

Every node can store data in stable storage, and this data can be obtained after a failure by the correct nodes. Nodes fail according to the fail-stop model. In this model, a node affected by a fault stops its execution and remains stopped until recovery is initiated. All correct nodes can determine which nodes have failed.

Each node provides a computational environment for one or more processes. Each process executes a program and exchanges messages with the other processes. Messages are delivered in any order (no FIFO requirement) and communication channels can be unreliable, i.e., channels can lose or duplicate messages. However, if channels are unreliable, a simple mechanism based on sequence numbers and timeouts can be used to guarantee that a process receives all messages sent to it without duplicates. Messages are delivered to processes with a *bounded delivery time*, t_{dmax}. The total time to send a message, transmit the message through the network, and then receive the message is smaller than t_{dmax}.

2.2 RECOVERABLE CHECKPOINTS

A distributed application is executed by a set of processes that run on several nodes. The main responsibility of a coordinated checkpoint protocol is to save global states of the application. A global state includes the state of each process belonging to the application and possibly some messages. A process state *contains* the send event $send(m_i)$ if it has sent message m_i. A process state contains the receive event $rcv(m_i)$ if it has received message m_i. A generic coordinated protocol should record *recoverable consistent global states*, which satisfy the following two properties:

Consistency : If the global state includes a process state containing the receive event $rcv(m_i)$ then another process state must contain the corresponding send event $send(m_i)$.

Recoverability : If the global state includes a process state containing the send event $send(m_i)$ but no other process state contains the corresponding receive event $rcv(m_i)$ then the checkpoint protocol must save message m_i.

Global states saved by the checkpoint protocol are used to recover the application from failures that have affected one or more of its processes. Typically, the application rolls back to the last stored state and then starts to re-execute. The external results of the application re-execution should be equivalent to one of the results of a failure-free execution. This can only be accomplished if the application restarts from a global state that could have occurred during one execution with no failures (Chandy and Lamport, 1985). For this reason, the global state can only contain receive events whose corresponding send events are also included. This characteristic is guaranteed by the consistency property. On the other hand, global states must include all messages that were in-transit at checkpoint time. Otherwise, these messages become lost during recovery because they are not re-sent by the processes. The recoverability property guarantees that all in-transit messages are available after the failure.

3. TIME-BASED CHECKPOINTING

The time-based checkpoint protocol of this chapter uses an *initialization procedure* to synchronize checkpoint timers and a *checkpoint creation procedure* to record recoverable consistent states of the application. The checkpoint creation procedure is executed periodically by each process whenever the local checkpoint timer expires. All processing is done locally without any exchanges of coordination messages. To guarantee that the consistency property is verified, the protocol disallows message sends during an interval after the expiration of the checkpoint timer. This interval is not constant, and increases as clocks drift apart. In an actual implementation, the blocking of message sends should not bring performance losses, because each process uses the interval to save its state. Timers are resynchronized when the interval becomes higher than the time taken to store a checkpoint. The recoverability property is ensured by preventing in-transit messages from occurring. The protocol disallows message sends during an interval before the checkpoint time. This interval is proportional to the maximum message delivery time.

```
Initialization:
  Coordinator:
    $ckpTime$ = getTime() + $T$;
    setTimer($createNewCkp, ckpTime$);
    setTimer($stopSMesg, ckpTime - (D + 2T\rho + t_{dmax})$);
    while (TRUE) {
      $time$ = getTime();
      broadcast($ckpTime - time$);
      for each($p_i \in Processes$) do receive($p_i$);
      if ((getTime() $- time) < (2 * t_{dmin} + D)$) {
        broadcast(FALSE);
        break;
      } else broadcast(TRUE);
    }
  Process i:
    $continue$ = TRUE;
    while ($continue$) {
      receive($coord, interval$);
      $time$ = getTime();
      send($coord$);
      receive($coord, continue$);
    }
    $ckpTime = time + interval - t_{dmin}$;
    setTimer($createNewCkp, ckpTime$);
    setTimer($stopSMesg, ckpTime - (D + 2T\rho + t_{dmax})$);
```

Figure 14.1 Initialization procedure.

3.1 INITIALIZATION PROCEDURE

The initialization procedure starts the processes' checkpoint timers in such a way that timers will expire within an interval of D seconds (if $\rho = 0$). Ideally, D should be made as small as possible, because that reduces the periods in which processes are not allowed to send messages (see next section). The initialization procedure is executed in three situations: to initialize the checkpoint protocol when the application starts, to initialize new processes that are added during the application execution, and to resynchronize the checkpoint timers. The resynchronization frequency depends on the value of the drift rate, but is relatively small.

The initialization procedure selects one of the processes to be the coordinator (the coordinator is usually the process that starts the application). The responsibility of the coordinator is to initiate the timers of the other processes. The coordinator cannot send an absolute time to the other processes, because clocks are not synchronized. It has to send a time interval. To calculate the interval

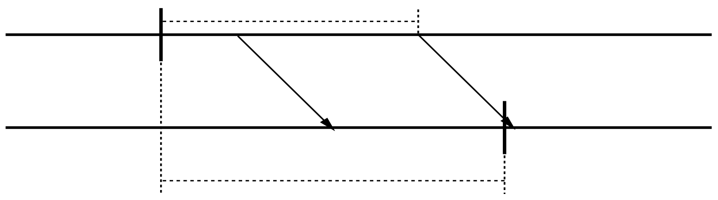

Figure 14.2 Consistency problem.

$inter_c$, the coordinator subtracts its local time from the time when its timer expires. The timer at process p_i is set to $timer_i = currentTime_i + inter_c - t_{dmin}$ (where t_{dmin} is the minimum time to deliver a message).

There are several ways to distribute the time interval among the processes, and their complexity depends on the system that is being considered. Figure 14.1 shows one implementation of the initialization procedure. First, the coordinator adds to its local time the checkpoint period T^1 to obtain the first checkpoint time, $ckpTime$. Then, it sets two timers that will call the functions $createNewCkp$ and $stopSMesg$ (the next section explains the time values that were used), and broadcasts the interval. The other processes execute the code `Process i`. This code receives the interval and initiates two similar timers. Since different messages can experience distinct network delays, the coordinator loops sending the interval until it receives all answers within a time period smaller than $D + 2 * t_{dmin}$. This guarantees that checkpoint timers will expire at most D seconds apart (if $\rho = 0$). In our experiments, D was set to 10 ms, which in most cases allowed the initialization of the timers in a single iteration.

3.2 CHECKPOINT CREATION

3.2.1 Consistency. The checkpoint creation procedure has to save application states that verify the consistency property. Figure 14.2 shows an example of an execution that violates the consistency property. Process Pi saves its state whenever its checkpoint timer expires at Ti. Since timers are not exactly synchronized, process $P1$ sends a message $m1$ after saving its state, and $P2$ receives $m1$ before storing its state. The consistency property is violated because the global state contains $recv(m1)$ but does not include $send(m1)$. To avoid this problem, the time-based protocol disallows message sends during an interval after the checkpoint timer has expired.

Consistency condition : The nth application checkpoint satisfies the consistency property if no process is allowed to send messages $MD - t_{dmin}$ seconds after saving its nth checkpoint.

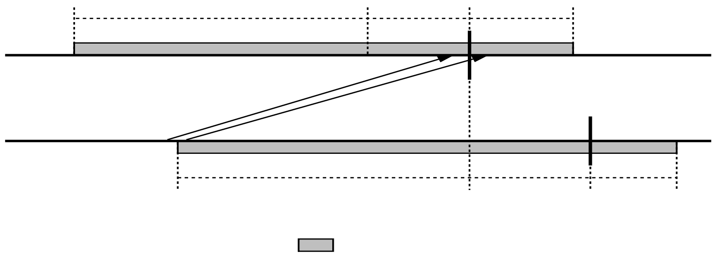

Figure 14.3 Total blocking interval.

The consistency condition (CC) defines an interval in which processes can not send messages. This interval is equal to the *maximum deviation*, MD, between timers minus the minimum time required to deliver a message. Timers in different processes do not expire at the same time, because they are not exactly synchronized. The maximum deviation is the maximum time interval that can separate the expiration of any two timers, and is equal to $MD = D + 2nT\rho$. It depends on two quantities, the initial deviation between timers, D, and the clock drift since the last initialization, $2nT\rho$. The first quantity is constant, but the second one increases with time. This means that the amount of blocking can grow with time. However, MD increases slowly because drift rates are small. For instance, the initialization procedure can be used to start timers with $D = 10$ ms. If we assume a clock drift of $\rho = 10^{-6}$, then MD is equal to 100 ms after 12.5 hours.

The performance losses introduced by the CC condition are usually small in real systems. The CC condition does not prevent processes from continuing their execution. The CC condition only blocks a process if it attempts to send messages (actually, the process only has to block if it sends a synchronous message, because asynchronous messages can be queued). Also, the blocking interval can be used to save the processes' state. In current systems, disk accesses consume a large amount of time, which means that most or all MD time is used to store the process checkpoint (a typical process checkpoint takes at least 500 ms).

3.2.2 Recoverability. The easiest way to guarantee the recoverability property consists of avoiding the creation of messages that might become in-transit. This approach simplifies the implementation of the protocol during both the failure-free periods and recovery. The checkpoint protocol does not need to log any messages or to re-send or re-read messages. However, in this solution, processes can not send messages during an interval before their timers expire.

```
stopSMesg()
  $stopSendMesg$ = TRUE;
  setTimer($stopSMesg, ckpTime + T-$
              $(D + 2(CN + 1)T\rho + t_{dmax})$);
createNewCkp()
  saveProcessState();
  $CN$ = $CN + 1$;
  $ckptTime$ = $ckpTime + T$;
  setTimer($createNewCkp, ckpTime$);
  if ((getTime() $-$ $(ckpTime - T)$) $\leq$
                 $(D + 2(CN - 1)T\rho - t_{dmin})$)
    resynchronizeTimers();
  $stopSendMesg$ = FALSE;
  sendQueuedMessages();
```

Figure 14.4 Checkpoint creation procedure.

Recoverability condition : The nth application checkpoint satisfies the recoverability property if no process is allowed to send messages $MD + t_{dmax}$ seconds *before* its timer expires.

The example from Figure 14.3 can be used to illustrate the recoverability condition (RC). A message can become in-transit if it is sent before a process creates its checkpoint and is received after the other process has saved its state. In the figure, process $P2$ sends two messages, $m1$ and $m2$. Message $m1$ does not have to be stored, but message $m2$ would have to be saved if process $P2$ was allowed to send it. The maximum interval that prevents the existence of messages like $m2$ is equal to $MD + t_{dmax}$. The reader should note that RC does not prohibit processes from continuing their executions until they start to save their checkpoints. The process needs to block only if it attempts to send a synchronous message. If the message can be sent asynchronously, the process simply queues the message and continues with the computation. The message is sent after CC is verified.

3.2.3 Creation of a Checkpoint. The time-based checkpoint protocol uses the CC and the RC conditions to create recoverable consistent checkpoints. The protocol can be implemented using the initialization procedure from Figure 14.1 and the checkpoint creation procedure from Figure 14.4. The creation procedure uses two timers: one that expires $MD + t_{dmax}$ seconds before checkpoint time, and another that expires at checkpoint time. Whenever the first timer terminates, it calls the function `stopSMesg`. This function sets a flag indicating that messages should be queued, and resets the timer. The function `createNewCkp` saves the process state, increments the checkpoint time by the checkpoint period T, and re-sets the timer. The variable CN counts the number of checkpoints that have been created since the last resynchronization. Then, `createNewCkp` tests the CC condition. If the condition is not verified, the function `resynchronizeTimers` is called to resynchronize the timers. This function sends a request for synchronization to the coordinator. The resynchronization procedure is similar to the initialization. Before returning, `createNewCkp` resets the flag and sends the messages that were queued.

4. MOBILE COMPUTING

4.1 UNIQUE ASPECTS OF MOBILITY

Mobile computing enables users to access and exchange information while they travel, roam in their home environment, or work at a client's site. Currently, mobile computing can only be used in restricted contexts; however, the growing investment by industry, researchers, and users indicates that the capabilities and applications of mobile computing will significantly increase (Forman and Zahorjan, 1994) (Nemzow, 1995) (Perkins, 1997).

Mobile hosts have a variety of computational and networking capabilities. For instance, pagers mainly serve to receive or send small messages. Personal digital assistants can have more sophisticated applications, such as an electronic organizer, and in the future will be able to receive and send external information such as airline schedules and reservations. Portable computers already provide computational power comparable to fixed hosts. These devices can execute general applications such as editors, spreadsheets, and databases. Portable computers can also have flexible networking capabilities, which allow them to connect either to hard-wired or wireless networks.

Wireless networking is useful in environments where hard-wired networks are not feasible or economically rewarding. Temporary networks can also be built faster and in a more cost-effective way by using wireless instead of hard-wired LANs. This quality is particularly interesting for disaster recovery after a fire, flood or earthquake (Nemzow, 1995). Currently several vendors are selling hardware support for wireless communication, using technologies like infrared transmitters and cellular telephone systems.

Mobile hosts have several characteristics that make them different from fixed hosts. Checkpoint protocols designed for mobile environments should consider these distinguishing features in their definition. Otherwise, they will incur high overheads, or they will simply not work correctly.

1. *Location is not fixed*: As the user moves from one place to another, the location of the mobile host in the network changes. The checkpoint protocol can store the processes' states in a well known site or in a computer near the current location of the mobile host. In the second

case, the checkpoint protocol has to keep track of the places where processes' states were saved.

2. *Disconnection*: A mobile host becomes disconnected when it goes outside the transmitting range of the emitters. While disconnected, the mobile host is not able to send or receive any messages. Protocols that need to exchange messages will not work correctly in this situation. During disconnection, the checkpoint protocol should provide a local recovery mechanism that allows the mobile host to recover from its own failures.

3. *Batteries store a limited amount of power*: The mobile host is often powered by batteries. Network transmissions and disk accesses are two of the most important sources of power consumption (Forman and Zahorjan, 1994). To minimize power consumption, the checkpoint protocol should reduce the amount of information that it adds to the application's messages, and it should avoid sending extra messages. The protocol should also make a small number of accesses to disk.

4. *Network characteristics are not constant*: The various wireless technologies have completely different qualities of service (Nemzow, 1995). For instance, typical radio frequency LANs currently have bandwidths between 2 and 20 Mbps, but a wide-area LAN using cellular digital packet data may have a bandwidth of 19.2 Kbps. Other different characteristics are cost, packet loss rates, and latency. The checkpoint protocol should adapt its behavior to the current network.

5. *Different types of failures*: Mobile host failures can be separated into two different categories. The first one includes all failures that can not be repaired; for example, the mobile host falls and breaks, or is lost or stolen. The second category contains the failures that do not permanently damage the mobile host; for example, the battery is discharged and the memory contents are lost, or the operating system crashes. The first type of failures will be referred to as *hard failures*, and the second type as *soft failures*. The protocol should provide different mechanisms to tolerate the two types of failures.

4.2 MOBILE IP

The mobile environment model used in this chapter is based on the mobile IP protocol (Perkins, 1997). The system contains both fixed and mobile hosts interconnected by a backbone network (see Figure 14.5). A mobile host uses a wireless interface to maintain network connections while it moves, and it is identified by a long term address. The address also serves to localize the mobile

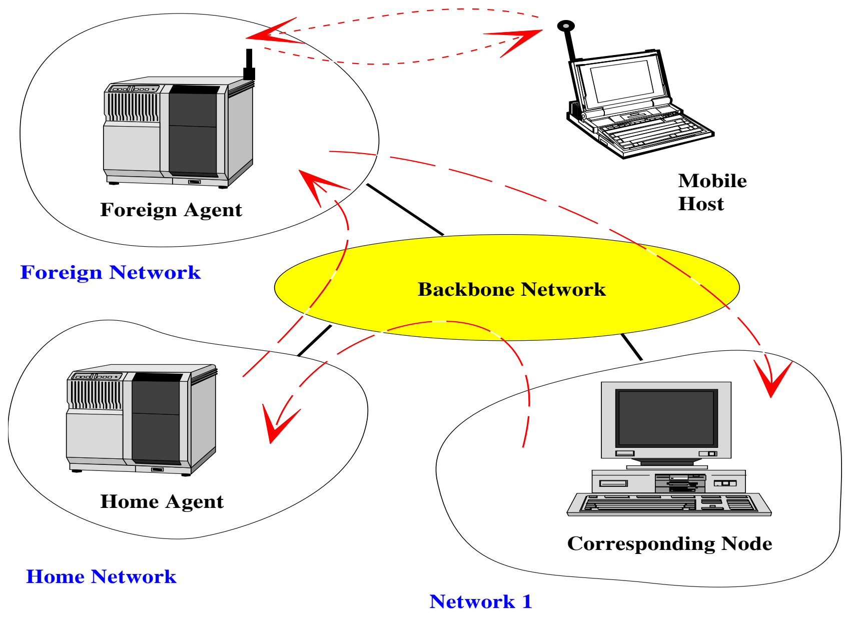

Figure 14.5 Mobile environment.

host's *home network*. While at home, the mobile host receives the packets as a normal fixed host. When it moves to another network, the mobile host relies on the services of a *foreign agent* to be able to communicate. Typically, the foreign agent has a wireless interface and is able to forward packets to and from the mobile host (the mobile host can also be directly connected to the wired network). The geographical cover area of the wireless interface is called the *cell*. Disconnection occurs when the mobile host moves outside the range of all the cells. The *home agent* represents the mobile host when it is away from the home network. The home agent intercepts the packets directed to the mobile host and forwards them to the current foreign agent[2]. The home node is informed by the mobile host about foreign agent changes.

The example from Figure 14.5 can be used to illustrate the communication between the mobile host and another host. The *corresponding host* sends packets to the long term address of the mobile host. These packets are routed by the backbone network to the home network. The routing protocol is the same as for packets that are sent to a fixed host. On the home network, the home agent intercepts the packets and forwards them to the foreign agent. The foreign agent transmits the packets through the wireless network to the mobile node. Packets sent by the mobile node do not have to be forwarded by the home agent. The foreign agent sends the mobile host's packets directly to the corresponding node.

```
// $S$ = Sender's identifier
// $CN$ = Current checkpoint number of the sender
// $timeToCkp$ = Time interval until next checkpoint
// $mesg$ = Message contents
receiveMesg($S, CN, timeToCkp, mesg$)
  if (($CN\_local == CN$) and (timeToCkp() $>$ $timeToCkp$))
    resetTimer($timeToCkp$);
  else if ($CN\_local < CN$) {
    createCkp();
    resetTimer($timeToCkp$);
  }
  deliverMesgToApplication($mesg$);
```

Figure 14.6 Message reception.

5. ADAPTIVE MOBILE RECOVERY

The adaptive checkpoint protocol uses time to indirectly coordinate the creation of global states. Processes save their states periodically, whenever a local checkpoint timer expires. The protocol can set different checkpoint intervals to ensure distinct recovery times. Higher checkpoint intervals require on average larger periods of re-execution, but reduce the protocol's overheads.

The protocol creates two distinct types of checkpoints (Neves and Fuchs, 1997a). The protocol uses checkpoints saved locally in the mobile host to tolerate soft failures, and it uses checkpoints stored in stable storage to recover hard failures. The first type of checkpoint is called *soft checkpoints*, and second type *hard checkpoints*. Soft checkpoints are necessarily less reliable than hard checkpoints, because they can be lost with hard failures. However, soft checkpoints cost much less than hard checkpoints because they are created locally, without any message exchanges. Hard checkpoints have to be sent through the wireless link, and then through the backbone network, until they are stored in stable storage.

5.1 TIME-BASED CHECKPOINTING

As described earlier for general distributed systems, the adaptive protocol uses time to avoid having to exchange messages during the checkpoint creation. A process saves its state whenever the local timer expires, independently from the other processes. The protocol keeps the various timers roughly synchronized to guarantee that processes' states are stored approximately in the same instant. When the application starts, the protocol sets the timers in all processes with a fixed value, the *checkpoint period*. The protocol uses a simple re-synchronization mechanism to adjust timers during the application execu-

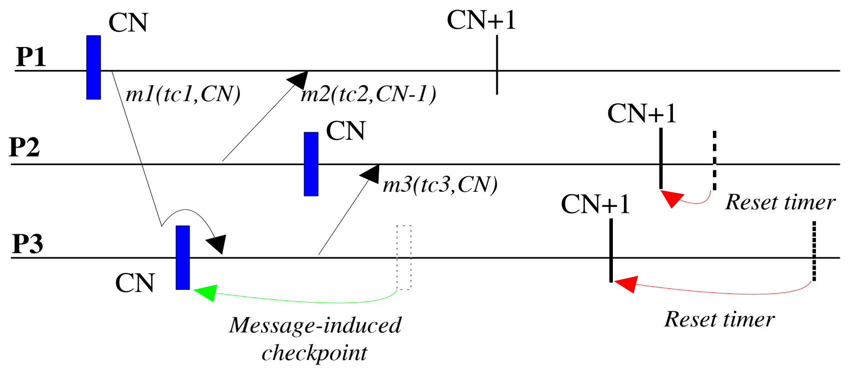

Figure 14.7 Time-based checkpointing.

tion. Each process piggybacks in its messages the time interval until the next checkpoint. When a process receives a message, it compares its local interval with the one just received (see Figure 14.6). If the received interval is smaller, the process resets its timer with the received value. The re-synchronization mechanism serves to solve initial timer inaccuracies and other causes of timer incorrections, such as clock drifts.

The protocol maintains a *checkpoint number* counter, *CN_local*, at each process to guarantee that the independently saved checkpoints verify the consistency property. The value of *CN_local* is incremented whenever the process creates a new checkpoint, and is piggybacked in every message. The consistency property is ensured if no process receives a message with a CN larger than the current local *CN_local*. The process creates a new checkpoint before delivering the message to the application if CN is larger than the local *CN_local* (see Figure 14.6). The recoverability property is guaranteed by logging at the sender all messages that might become in-transit. These are the messages that have not been acknowledged by the receivers at checkpoint time. The sender process also logs the send and receive sequence number counters. During normal operation, these counters are used by the communication layer to detect lost messages and duplicate messages due to retransmissions. After a failure, each process re-sends the logged messages. Duplicate messages are detected as they are during the normal operation.

The example from Figure 14.7 will be used to illustrate the execution of the protocol. This figure represents the execution of three processes (to simplify the figure, message acknowledgments are not represented). Processes create their checkpoints at different instants, because timers are not synchronized. After saving its CN checkpoint, process $P1$ sends message $m1$. When $m1$ arrives, process $P3$ is still in its $CN - 1$ checkpoint interval. To avoid a consistency problem, $P3$ first creates its CN checkpoint, and then delivers $m1$. $P3$ also

Table 14.1 Configuration Table for maxSoft.

Quality of Service	*maxSoft*		*Network Example*
	Low	*High*	
$QoS > 10$	1	2	ethernet, ATM
$6 < QoS \leq 10$	2	8	radio, infrared
$3 < QoS \leq 6$	4	32	cellular
$0 < QoS \leq 3$	8	128	satellite
$QoS = 0$	∞	∞	disconnected

resets the timer for the next checkpoint. Message $m2$ is an in-transit message that has not been acknowledged when process $P2$ saves its CN checkpoint. This message is logged in the checkpoint of $P2$. Message $m3$ is a normal message that indirectly re-synchronizes the timer of process $P2$. It is possible to observe in the figure the effectiveness of the re-synchronization mechanism.

5.2 SOFT VS. HARD CHECKPOINTS

The protocol adapts its behavior to the characteristics of the network. For instance, if the network has a poor quality of service, the protocol saves many soft checkpoints before it sends a hard checkpoint to stable storage. The number of soft checkpoints that are stored per hard checkpoint is called $maxSoft$, and it depends on the quality of service of the current network. The assignment of $maxSoft$ values to the different networks is made statically, and saved in a table. Table 14.1 gives two examples of possible assignments. The minimal quality of service corresponds to a disconnected mobile host. In this case, $maxSoft$ is set to infinity, which means that only soft checkpoints are created. The low $maxSoft$ column represents an assignment where hard checkpoints are created frequently, which guarantees a small re-execution time after a hard failure. The high $maxSoft$ column corresponds to the opposite case.

Application processes run on hosts that might be connected to different networks, each corresponding to a distinct $maxSoft$ value. This means that a global state can include both soft and hard checkpoints. To ensure that recovery is always possible, the protocol has to keep at each moment a global state containing only hard checkpoints. This global state is used to recover the application from hard failures. Otherwise, the domino effect can occur, and recovery might not be possible. The protocol guarantees that new hard global states are saved by correctly initializing the $maxSoft$ table. The process that creates hard checkpoints less frequently is the one running in the host connected to the network with worse quality of service (we will discuss the disconnect case in the next section). The protocol guarantees that a new hard global state

```
Application process:
createCkp()
  $CN\_local = CN\_local + 1$;
  resetTimer($T$);
  if (($CN\_local mod maxSoft$) == 0) sendCkpST(getState());
  else storeState(getState(), $CN\_local$);

Stable storage:
// The function arguments are the same as in receiveMesg()
receiveCkp($S, CN, timeToCkp, state$)
  $CN\_local = \max(CN\_local, CN)$;
  setBit($CN, S$);
  if (row($CN$) == 1) {
    $CN\_hard = CN$;
    garbageCollect($CN\_hard$);
  }
```

Figure 14.8 Functions to create a new checkpoint.

is stored every time this process creates a hard checkpoint, by initializing the table in such a way that $maxSoft$ values are multiples of each other. For example, if processes $P1$ and $P2$ have $maxSoft$ values 4 and 8, this means that a new hard global state is stored every 8 checkpoints. Process $P1$ creates hard checkpoints whenever *CN_local* is equal to 4, 8, 12, 16, ..., and process $P2$ whenever *CN_local* is equal to 8, 16, ... The protocol also keeps the last global state that was stored (which can include soft checkpoints) to recover from soft failures.

The functions from Figure 14.8 are used to create a new checkpoint. Function `createCkp` is called to save a new process state. It starts by incrementing the *CN_local*, and then it resets the timer with the checkpoint period. Next, the function determines if the checkpoint should be saved locally or sent to stable storage. The function `storeState` stores locally the process state, and the function `sendCkpST` sends the process state to stable storage. The function `receiveCkp` is called by the stable storage to store newly arrived checkpoints. It first writes the received state to the disk, and then updates the local checkpoint counter. Then, it determines if a new hard global state has been stored using a *checkpoint table*. The checkpoint table contains one row per CN, and one column per process. The table entries are initialized to zero. An entry is set to one whenever the corresponding checkpoint is written to disk. The table only needs to keep one bit per entry, which means that it can be stored compactly. A new hard global state has been saved when all entries of a row are equal to one. The variable *CN_hard* keeps the checkpoint number of the new hard global state. The function `garbageCollect` removes all checkpoints with checkpoint numbers smaller than *CN_hard*.

5.3 MOBILE HOST DISCONNECTION

A mobile host becomes disconnected whenever it moves outside the range of all the cells, or whenever the user turns off the network interface. While disconnected, the mobile host can not access any information that is stored in the stable storage. For this reason, the protocol must be able to perform its duties correctly using only local information. The protocol continues to save soft checkpoints in order to recover from soft failures. We consider two different types of disconnection. An *orderly disconnection* allows the protocol to exchange a few messages with the stable storage just before the mobile becomes isolated. Examples of this type of disconnection include situations in which the user calls a logout command, or the communication layers inform the protocol when the mobile is about to move outside the range of the cells (when the wireless signal becomes weaker). A *disorderly disconnection* corresponds to the opposite case, in which the protocol is not able to exchange any messages with stable storage. This happens, for instance, when the user unplugs the ethernet cable without turning off the application (Neves and Fuchs, 1997b).

The creation of a new global state before disconnection is advantageous for both the mobile host and the other hosts. This new global state is important because it prevents the rollback of work that was done while the mobile host was disconnected. If the new global state is not saved and another host fails after the disconnection, the application rolls back to the last global state that was stored (without warning the mobile host). Later, during re-connection, the mobile's process will be warned about the failure and will also have to rollback, undoing the work executed during the disconnection. The same principle can be applied to the failures of the mobile host and the work done by the other hosts.

The mobile host cooperates with the stable storage to create a new global state before disconnection. Just before the mobile host becomes isolated, the protocol sends to stable storage a request for checkpoint, and saves a new checkpoint of the process (hard or soft, depending on the network). Then, the stable storage broadcasts the request to the other processes. Processes save their state as they receive the request. New global states can only be created before the mobile host detaches from the network if disconnections are orderly. Otherwise, the protocol is not able to determine when disconnections occur. In any case, the protocol can always create a local checkpoint. This soft checkpoint allows independent recovery of soft failures, and minimizes the probability of global rollbacks due to failures of the mobile host.

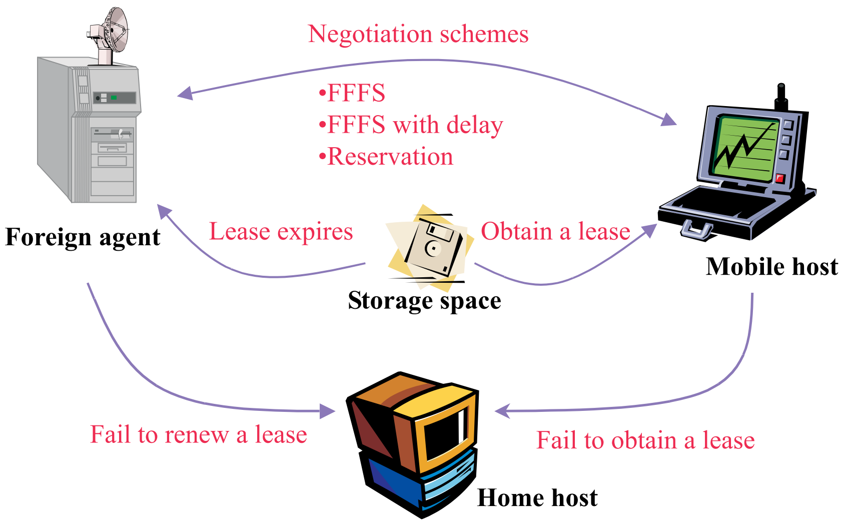

Figure 14.9 Leasing for the foreign agent.

When the mobile host re-connects, the protocol sends a request to stable storage, asking for the current checkpoint number and the CN of the last hard global state. When the answer arrives, the protocol updates the local *CN_local* using the current checkpoint number. The protocol also creates a hard checkpoint if the mobile host has been isolated for a long time. If the difference between *CN_local* and *CN_hard* is larger than the maximum $maxSoft$ (in the example from Table 14.1, 8 or 128 depending on the assignment), the mobile sends a new hard checkpoint to stable storage. This checkpoint allows the hard global state to advance.

6. FOREIGN AGENTS

6.1 LEASING AND STORAGE

Some foreign agents provide temporary storage service for mobile users. The concept of leasing can be used for managing stable storage on foreign agents (Yin et al., 1998). With information provided through leasing, the storage manager knows the exact minimum available storage at any specific time and thus is able to appropriately arrange for future space utilization. Leasing can also prevent storage resources from being held indefinitely by failed or blocked processes. Both the process and manager know the expiration time of a lease and thus garbage collection is simplified.

The leasing mechanism of this chapter is described as follows (see Figure 14.9). Every process that needs to utilize stable storage negotiates with the manager for the size of the space and the length of the lease. As the lease expires, the process must either obtain a lease extension (new lease) or the space is returned to the manager. The amount of space and the length of a new lease may vary from the original lease. The storage manager may either grant or decline the renewal based on the management protocol. The leasing mechanism has the following four properties:

Negotiation: The storage manager and the process negotiate the expiration time of the lease and the size of the storage. The lease is valid only when the manager and process both agree to the lease.

Cancellation: The process can cancel the lease and return the space to the storage manager at any time before the lease expires. The manager, however, does not have the right of cancellation.

Renewal: The process has the right to request a new lease before the expiration time of the lease. The storage manager may either grant or decline the request based on the storage management policy.

Expiration: Every lease has an expiration time. The process must return the storage to the manager if the lease is not successfully renewed.

6.2 ADAPTIVE CHECKPOINTING

Our approach to adaptive checkpointing with leasing uses time to indirectly coordinate the creation of the checkpoints and it utilizes a three-level storage hierarchy to save the checkpoints. This chapter previously described how time can be an efficient mechanism for implementing mobile checkpointing. This section describes how leasing can be integrated with adaptive time-based checkpointing to enhance the performance of hierarchical storage management.

The protocol uses a three-level storage hierarchy to save the checkpoints of the processes. Checkpoints stored in the first level are *soft checkpoints (SC)*, and they are saved in the mobile host (e.g., in a local disk or flash memory). The other two levels correspond to the stable storage available in the foreign agents and home host and are *hard checkpoints (HC)*. Soft checkpoints are less reliable than hard checkpoints because they can be used to tolerate only temporary failures of the mobile host. The hard checkpoints are able to survive permanent failures but have higher overheads due to their transmission through the wireless channels. Based on the quality of service of the current network, the protocol selects a specified ratio between soft and hard checkpoints for the best reliability and performance. For example, it can send a hard checkpoint to the stable storage whenever a fixed number of soft checkpoints have been created in the local disk.

There are distinct space requirements throughout the storage hierarchy. In the mobile host it is only necessary to save a soft checkpoint for the process

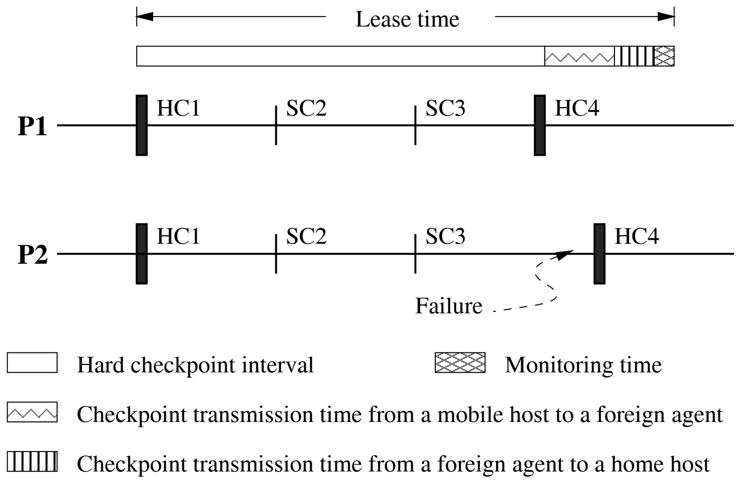
Lease time
P1
HC1
SC2
SC3
HC4
P2
HC1
SC2
SC3
HC4
Failure
Hard checkpoint interval
Monitoring time
Checkpoint transmission time from a mobile host to a foreign agent
Checkpoint transmission time from a foreign agent to a home host

Figure 14.10 Leasing time.

locally executing the application. The stable storage in a foreign agent has to be shared among the mobile hosts currently in the cell. These mobile hosts may execute different applications with distinct checkpoint intervals and sizes. Therefore, the foreign agents use the leasing mechanism to manage the stable storage. The home host retains global states of the application. A global state contains one checkpoint per each process executing the application. We assume that there is enough space to store the checkpoints in the mobile and home hosts. This assumption is reasonable since these hosts likely belong to the same organization, which means that they can be configured to support the storage requirements of the user applications.

The protocol first attempts to save the hard checkpoints in the foreign agents instead of the home host due to performance advantages. The failure free performance is better because one transmission step is avoided. A checkpoint has first to pass through the foreign agent before it is sent to the home host. Moreover, recovery is faster because checkpoints are closer to the hosts. Storing checkpoints in the foreign agents, however, raises problems that have to be addressed by the checkpoint protocol. For instance, since the timers may not be well synchronized, a permanent failure can occur during the time when some processes have completed their checkpoints while others are in progress (see *HC4* in Figure 14.10). If the failure is detected before the termination of the leases on the previous checkpoints that form a consistent recovery line (*HC1* of *P1* and *P2*), then recovery can be achieved. Otherwise, the protocol will not have a consistent set of checkpoints for recovery. Requests for storage sometimes may not be immediately granted if there is no sufficient available space in the foreign agent. In this case, the protocol has to either postpone the hard checkpoint or save it in another location.

The protocol negotiates with the foreign agents and home host to determine the location to save the hard checkpoints. Whenever it is time to store a new hard checkpoint, the process contacts the local foreign agent and tries to obtain a lease for the required space. Then, it sends the checkpoint through the wireless link and transmits a completion notification to the home host. If it is impossible to obtain a lease within an allowable delay, the process stores the checkpoint directly in the home host. At this moment, the process has finished the checkpoint creation. On the home host, a monitoring process is initiated after arrival of the first completion notification. The monitoring process ensures that a new global state is saved in stable storage before the previous checkpoint is garbage collected by the storage manager. The monitor expects to receive a notification from all processes within a given *monitoring time*, otherwise it assumes that a failure may have occurred. In this case, the monitor requests from the foreign agents a copy of the previous checkpoint and saves them in the local stable storage. This operation guarantees that there is a complete global state available for recovery.

The lease time must ensure that the current hard checkpoint of the process will be safely stored in the foreign agent until the next hard checkpoint is created. Moreover, it has to be sufficiently large to allow the home host to collect the checkpoint copies in case of failures. Therefore, the lease time is set to be the sum of the hard checkpoint interval and the extra time that is the monitoring time and the time to transmit the checkpoint from the mobile host to the home host (see Figure 14.10). With this establishment of the lease time, at least one consistent global state can be preserved. With failure-free execution, the global state will typically have been created before the leases expire. The monitoring process can send lease termination requests once all the notifications have been received.

6.3 HAND-OFF PROCEDURES

Before moving to another cell, the process notifies the storage manager of the current foreign agent. The manager then forwards the hard checkpoint(s) of the process to the home host. After the checkpoint is saved safely by the home host, the checkpoint on the foreign agent is removed. If the new cell provides storage service and the process gets a lease, the hard checkpoint can alternatively be sent to the new foreign agent. This hand-off procedure simplifies the garbage collection on foreign agents. When the mobile host leaves the current cell, the space occupied by its checkpoints will be available for reallocation. This feature avoids having checkpoints scattered throughout the network as the mobile host moves around. The mobile host also does not have to maintain extra links to locate previous checkpoints.

7. SUMMARY

A checkpoint protocol was described that uses time to avoid performance penalties introduced by traditional coordinated protocols. The protocol does not rely on synchronized clocks to eliminate the message coordination overhead. It uses a simple initialization procedure to start the checkpoint timers. Contrary to previous time-based protocols, it also eliminates the overheads of in-transit message storage and addition of information to messages. This is accomplished by preventing processes from sending messages during an interval before the checkpoint time.

This chapter also described how the checkpoint protocol can be adapted to the characteristics of mobile environments. The protocol is able to save consistent recoverable global states without needing to exchange messages. As with general distributed systems, a process creates a new checkpoint whenever a local timer expires and a simple mechanism is used to keep the checkpoint timers approximately synchronized. The protocol saves soft checkpoints locally in the mobile host, and stores hard checkpoints in stable storage. The protocol adapts its behavior to different networks by changing the number of soft checkpoints that are created per hard checkpoint. When the mobile host is disconnected, the protocol creates soft checkpoints for recovery from soft failures.

The chapter demonstrated how adaptive checkpointing can be integrated with leasing. With this feature, processes that do not immediately obtain storage for necessary checkpoints are not forced to miss checkpoints. The protocol utilizes hierarchical storage management to improve checkpointing performance.

Acknowledgments

Portions of this chapter are from the following papers by the authors, Neves and Fuchs, "Adaptive Recovery for Mobile Environments," *Communications of the ACM*, vol. 40, no. 1, pp. 68–74, Jan. 1997, and "Using Time to Improve the Performance of Coordinated Checkpointing," *Proceedings of the International Computer Performance & Dependability Symposium*, pp. 282–291, Sept. 1996. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract DABT 63-96-C-0069, and in part by the Office of Naval Research under contract N00014-97-1-1013.

Notes

1. For simplicity, checkpoints are created periodically with a constant period T. In a more general case, T can be different for each checkpoint as long as processes agree on the same value.
2. Mobile IP also allows messages to be directly forwarded to the mobile host, if it has a temporary address belonging to the foreign network.

References

Acharya, A. and Badrinath, B. R. (1994). Checkpointing distributed applications on mobile computers. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 73–80.

Borg, A., Blau, W., Graetsch, W., Herrmann, F., and Oberle, W. (1989). Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75.

Cristian, F. and Fetzer, C. (1994). Probabilistic internal clock synchronization. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 22–31.

Cristian, F. and Jahanian, F. (1991). A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20.

Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W. (1992). The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47.

Elnozahy, E. N. and Zwaenepoel, W. (1992). Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531.

Elnozahy, E. N. and Zwaenepoel, W. (1994). On the use and implementation of message logging. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pages 298–307.

Forman, G. H. and Zahorjan, J. (1994). The challenges of mobile computing. *Computer*, 27(4):38–47.

Goldberg, A., Gopal, A., Li, K., Strom, R., and Bacon, D. (1990). Transparent recovery of Mach applications. In *Proceedings of the Usenix Mach Workshop*, pages 169–184.

Johnson, D. B. and Zwaenepoel, W. (1987). Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19.

Johnson, D. B. and Zwaenepoel, W. (1990). Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491.

Kim, J. L. and Park, T. (1993). An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):231–240.

Koo, R. and Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31.

Nemzow, M. (1995). *Implementing wireless networks*. McGraw-Hill Series on Computer Communications. McGraw-Hill, Inc., New York.

Neves, N., Castro, M., and Guedes, P. (1994). A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Systems*, pages 121–129.

Neves, N. and Fuchs, W. K. (1996). Using time to improve the performance of coordinated checkpointing. In *Proceedings of the International Computer Performance & Dependability Symposium*, pages 282–291.

Neves, N. and Fuchs, W. K. (1997a). Adaptive recovery for mobile environments. *Communications of the ACM*, 40(1):68–74.

Neves, N. and Fuchs, W. K. (1997b). Fault detection using hints from the socket layer. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, pages 64–71.

Neves, N. and Fuchs, W. K. (1998a). Coordinated checkpointing without direct coordination. In *Proceedings of the International Computer Performance & Dependability Symposium*, pages 23–31.

Neves, N. and Fuchs, W. K. (1998b). RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pages 58–67.

Perkins, C. E. (1997). *Mobile IP Design Principles and Practices*. Addison–Wesley.

Plank, J. S. (1993). *Efficient checkpointing on MIMD architectures*. PhD thesis, Princeton University.

Pradhan, D. K., Krishna, P., and Vaidya, N. H. (1996). Recovery in mobile environments: Design and trade-off analysis. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pages 16–25.

Ramanathan, P. and Shin, K. G. (1993). Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Transactions on Software Engineering*, 19(6):571–583.

Silva, L. M. and Silva, J. G. (1992). Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162.

Ssu, K. F., Yao, B., and Fuchs, W. K. (1999). An Adaptive Checkpointing Protocol to Bound Recovery Time with Message Logging. In *Proceedings of the 18th Symposium on Reliable Distributed Systems*.

Strom, R. E. and Yemini, S. (1985). Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226.

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

Wang, Y.-M. and Fuchs, W. K. (1992). Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 147–154.

Wang, Y.-M. and Fuchs, W. K. (1993). Lazy checkpoint coordination for bounding rollback propagation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 86–95.

Yao, B., Ssu, K. F., and Fuchs, W. K. (1999). Message logging in mobile computing. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 294–301.

Yin, J., Alvisi, L., Dahlin, M., and Lin, C. (1998). Using leases to support server-driven consistency in large-scale systems. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 285–294.