This paper appeared in proceedings of the IEEE High-Assurance Systems Engineering Workshop, October 1996.

Adaptive Recovery for Mobile Environments

Nuno Neves
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

W. Kent Fuchs
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285

Abstract

Mobile computing allows ubiquitous and continuous access to computing resources while the users travel or work at a client's site. The flexibility introduced by mobile computing brings new challenges to the area of fault tolerance. Failures that were rare with fixed hosts become common, and host disconnection makes fault detection and message coordination difficult. This paper describes a new checkpoint protocol that is well adapted to mobile environments. The protocol uses time to indirectly coordinate the creation of new global states, avoiding all message exchanges. The protocol uses two different types of checkpoints to adapt to the current network characteristics, and to trade off performance with recovery time.

1 Introduction

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 [1–5].

Mobile hosts have a variety of computational and networking capabilities. For instance, pagers mainly serve to receive or send small messages. Personal digital assistants (PDAs) 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 [6]. 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 [7]. Currently several vendors are selling hardware support for wireless communication, using technologies like infrared transmitters and cellular telephone systems.

The diversity and flexibility introduced by mobile computing bring new challenges to the area of fault tolerance. Types of failures that were rare in fixed environments are common with mobile hosts. Physical damage becomes much more probable, because mobile hosts are carried with the users while they roam between places. Mobile hosts can also be lost or stolen. Transient failures due to power or connectivity problems will be frequent events.

In this paper, we focus on checkpoint-based recovery techniques for distributed systems. A checkpoint protocol typically functions as follows. The protocol periodically stores the state of the application in stable storage. After a failure, the application rolls back to the last saved state and then re-starts its execution. Checkpoint protocols proposed in the past are not adequate for mobile environments (see Section 3). Coordinated protocols have to exchange messages during checkpoint creation to save recoverable consistent global states [8–11], which is something that should be avoided because of disconnections. Uncoordinated protocols can create checkpoints without having to communicate [12–15]. However, during recovery they may need to send messages to find global states [16], or to obtain information saved in the surviving processes [14, 17]. Previous protocols also have the problem that they do not adapt their behavior to the characteristics of the current network connection. If the network has a small bandwidth and a high failure rate, the protocol should be able to trade off recovery time with operational costs. The user may prefer to utilize

Nuno Neves was supported in part by the Grant BD-3314-94, sponsored by the program PRAXIS XXI, Portugal. This research was supported in part by the Office of Naval Research under contract N00014-95-1-1049, and in part by the Defense Advanced Research Projects Agency (DARPA) under contract DABT 63-96-C-0069. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

the available bandwidth with the application's messages, instead of using it to send the checkpoints to stable storage. The protocol should also give distinct treatment to different types of failures. For instance, rapid recovery should be provided for frequent failures.

This paper proposes a new coordinated checkpoint protocol for distributed systems. This protocol was designed to take into consideration the special characteristics of the 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 have 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. Process checkpoints are saved in stable storage or locally in the hosts. Locally stored checkpoints do not consume network bandwidth, and take much less time to be created. However, they can be lost due to permanent failures in the mobile hosts. During the application execution, the protocol keeps a global state in stable storage and has another global state that is dispersed through the mobile hosts and stable storage. The first global state is used to recover permanent failures, and the second transient failures.

2 Unique Aspects of Mobile Environments

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 can become disconnected. 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 [18]. 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 [3, 7]. For instance, a radio frequency LAN can have bandwidths between 2 and 20 Mbps, but a wide-area LAN using cellular digital packet data (CDPD) 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.

3 Related Work

Uncoordinated checkpoint protocols save the processes' states without having to exchange messages [12–15]. This is an interesting characteristic for mobile environments because processes can continue to create checkpoints while they are disconnected. On the other hand, these protocols usually have to save a reasonable amount of information to guarantee deterministic execution after a failure. This information includes the reception order and the contents of the messages. This can be a problem if the information has to be saved in a mobile host, since there is typically a limited amount of flash memory or disk. Uncoordinated protocols also have the problem that processes usually need to exchange messages to garbage collect the stored information [19]. During recovery, processes also send messages to find a consistent global state [16] or to obtain information stored by the other processes [14, 17].

Most of the coordinated checkpoint protocols exchange messages during the checkpoint creation [9–11]. Messages are needed to guarantee that processes' checkpoints form a consistent recoverable global state. This characteristic makes these protocols inadequate for mobile environments. Time-based coordinated protocols do not need to send the coordination messages [8, 20, 21]. However, they rely on synchronized clocks or timers, which is something that will be difficult to guarantee in mobile environments.

Two checkpoint protocols designed for mobile environments have been proposed [22, 23]. The protocol by Acharya and Badrinath [22] uses message-induced checkpointing to save consistent global states. Pradhan et al. [23] proposed an uncoordinated protocol that saves the state of the processes in the computer where a mobile host is currently attached (see Section 6 for a complete description of these protocols). Neither approach needs to exchange messages during the checkpoint creation. However, they may need to store a large number of checkpoints and/or messages. Also, they do not adapt to current characteristics of the network, and they always assume hard failures.

Vaidya has proposed previously a two-level scheme that uses two checkpoint protocols to tolerate distinct failures [24]. The two-level scheme relies on an uncoordinated protocol to tolerate single process failures, and on a coordinated protocol to recover multiple process failures. Our scheme utilizes a coordinated protocol that saves two different types of global states. The first type of global state is able to recover single or multiple soft failures, and the second type of global state is used to tolerate single or multiple hard failures.

4 Distributed System Model

4.1 Mobile Environment

The mobile environment model that is used in this paper is based on the current internet draft for mobile IP [25]. The system contains both fixed and mobile hosts interconnected by a backbone network (see Figure 1). 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 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 mobile host can request the services of another foreign agent if the current one fails. 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[1]. The home node is informed by the mobile

[1] Mobile IP also allows messages to be directly forwarded to the mobile host, if it has a temporary address belonging to the foreign network.

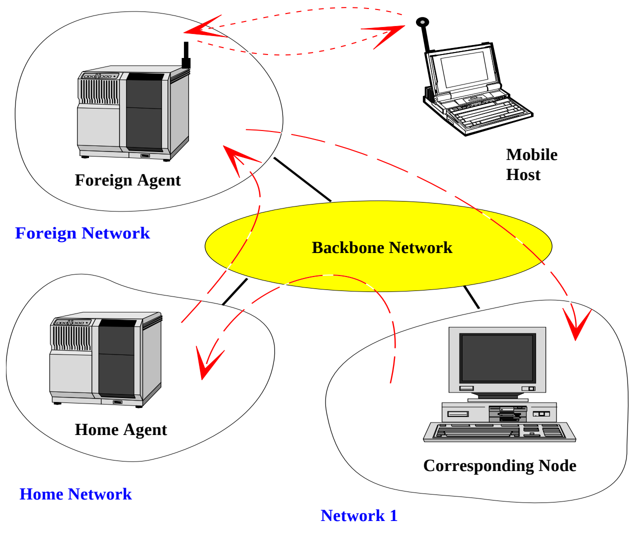

Figure 1: Mobile environment.

host about foreign agent changes.

The example from Figure 1 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.

4.2 Recoverable Consistent Checkpointing

A coordinated checkpoint protocol saves global states of an application that is executed by one or more processes. Processes run on fixed or mobile hosts, and use messages to exchange data. A global state includes the state of each process belonging to the application, and possibly some messages. Global states are used by the checkpoint protocol to recover the application from failures. Failure recovery is accomplished by rolling back the processes to the last stored state. Then, processes re-execute the application program and re-read the logged messages. Recovery is only correct if the external results of the application re-execution are equivalent to one of the results of a failure-free execution [26].

A correct recovery can only be guaranteed if the checkpoint protocol records *recoverable consistent global states*. These global states satisfy the following two properties:

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

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

The consistency property ensures that processes start their re-execution from a state that might have occurred on a failure-free execution. This can only be verified if all receive events reflected in the processes' states have the corresponding send events also reflected in the global state. The recoverability property guarantees that all in-transit messages at checkpoint time are included in the global state. Otherwise, these messages become lost during recovery, because they are not re-sent by the processes.

5 Adaptive Checkpoint Protocol

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. 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.

The protocol uses the distinct creation costs of the two checkpoint types to adapt its behavior to the quality of service of the current network. For different network configurations, the protocol saves a distinct number of soft checkpoints per hard checkpoint. If the network is slow, the protocol creates many soft checkpoints to avoid the network transmissions. By making a correct balance between soft and hard checkpoints, the protocol can keep its overheads approximately equal across the various types of networks.

For a given network configuration, the protocol can exchange hard failure recovery time with performance costs. Hard failures are recovered with global states containing

```
// $P_m$              = Sender's identifier
// $CN_m$             = Current checkpoint number of the sender
// $timeToCkp_m$      = Time interval until next checkpoint
// $mesg_m$           = Message contents
receiveMesg($P_m, CN_m, timeToCkp_m, mesg_m$):
   if (($CN = CN_m$) and (timeToCkp() $> timeToCkp_m$))
      resetTimer($timeToCkp_m$);
   else if ($CN < CN_m$) {      // orphan message
      createCkp();
      resetTimer($timeToCkp_m$);
   }
   deliverMesgToApplication($mesg_m$);
```

Figure 2: Message reception.

only hard checkpoints. If the protocol creates hard checkpoints frequently, the amount of rollback due to hard failures is small on average. However, the performance of the protocol can be poor.

Soft checkpoints let the protocol continue to function correctly while the mobile host is disconnected. Conceptually, a disconnected mobile host can be viewed as a host connected to a network with no bandwidth. In this case, the number of soft checkpoints per hard checkpoint is set to infinity, which means that all processes' states are stored locally. The local checkpoints are used to recover the mobile host from soft failures.

Recovery from soft failures should be faster than from hard failures, since soft failures will occur more frequently. This can be accomplished with soft checkpoints, because recovery can be made completely local, or at most it will be necessary to send a rollback request message to the other processes.

5.1 Time-Based Checkpointing

The adaptive protocol uses time to avoid having to exchange messages during the checkpoint creation. Previous protocols utilize a two-phase message coordination to determine when a new global state should be saved. The adaptive protocol accomplishes the same effect as message coordination by having the checkpoint timers at the different processes approximately synchronized. A process saves its state whenever the local timer expires, independently from the other processes.

The adaptive protocol uses two techniques to keep the timers roughly synchronized. When the application starts, the protocol sets the timers in all processes with a fixed value, the *checkpoint period*. Since processes do not begin to execute in exactly the same instant, timers will expire at different times. The protocol has a re-synchronization mechanism that adjusts timers during the application execution. Each process piggybacks in its messages the time interval until the next checkpoint. When a process receives

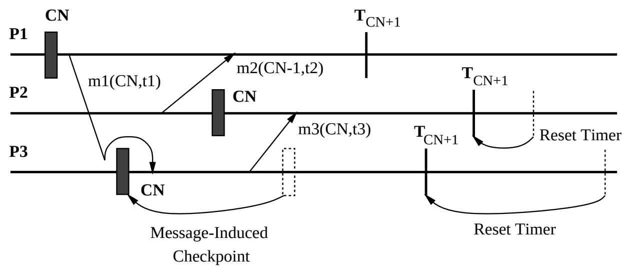

Figure 3: Time-based checkpointing.

a message, it compares its local interval with the one just received (see Figure 2). If the received interval is smaller, the process resets its timer with the received value. The re-synchronization mechanism also serves to solve other causes of timer incorrections, e.g., clock drifts.

The protocol maintains a *checkpoint number* counter, CN, at each process to guarantee that the independently saved checkpoints verify the consistency property [27, 28]. The value of CN 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_m larger than the current local CN. The process creates a new checkpoint before delivering the message to the application if CN_m is larger than the local CN (see Figure 2). 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 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 3 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 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. Timers are better synchronized after the three messages have been received.

Table 1: Configuration table for *maxSoft*.

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

5.2 Soft vs. Hard Checkpoints

The protocol adapts its behavior to the current characteristics of the network. For instance, if the network has a poor quality of service, the protocol saves several soft checkpoints before it sends a hard checkpoint to stable storage. The quality of service of a network depends on several factors, e.g., bandwidth and packet loss rate. Its value can be estimated by the protocol, or it can be provided by the underlying communication layers [1, 3].

The number of soft checkpoints per hard checkpoint is called $maxSoft$, and it changes as the mobile host moves from one type of network to another. If the new network has a better quality of service, the value of $maxSoft$ is decreased. The assignment of $maxSoft$ values to the different networks can be made statically, and saved in a table by the protocol. Table 1 gives two examples of possible assignments. The table presents the values of $maxSoft$ for each range of quality of service. 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.

The application is executed by several processes running on mobile and/or fixed hosts. Since the network configurations are different at the various hosts, processes save their states using distinct $maxSoft$ values. 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 [29] can occur, and recovery might not be possible. The protocol guarantees that a hard global state always exists, and that new hard global states are saved as the application continues its execution, by correctly initializing the $maxSoft$ table. A process saves a new hard checkpoint after creating $maxSoft$ soft checkpoints. The process that

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

// Stable storage:
// The function arguments are the same as in receiveMesg
receiveCkp($P_m, CN_m, timeToCkp_m, state_m$):
    storeState($state_m, CN_m$);
    $CN$ := max($CN, CN_m$);
    setBit($CN_m, P_m$);
    if (row($CN_m$) = 1) {
        $CN_{hard}$ := $CN_m$;
        garbageCollect($CN_{hard}$);
    }
```

Figure 4: Functions to create a new checkpoint.

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 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 is equal to 4, 8, 12, 16, 20, ..., and process $P2$ whenever CN is equal to 8, 16, 24, ... 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 4 are used to create a new checkpoint. Function `createCkp` is called to save a new process state. It starts by incrementing the CN, 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.

There are two reasons why the mobile host should create a new global state before it disconnects. From the mobile host owner's point of view, the protocol should create a new global state because it prevents the rollback of work done while the mobile was disconnected. If a failure occurs after the mobile's disconnection and before the creation of a new global state, 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 also have to roll back to this global state, undoing all the work executed while the mobile host was isolated. The creation of a new global state is also advantageous to the owners of the other hosts. If the mobile host (soft) fails after disconnecting and before saving its state, the protocol recovers the application process by doing a local rollback to the last checkpoint. When the mobile host re-connects, it will inform the other hosts about its failure (if the failure was hard, the user will have to tell the system administrator), and all the other processes will also have to roll back.

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 second problem that was mentioned in the previous paragraph.

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 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 the received CN and CN_{hard} is larger than the maximum $maxSoft$ (in the example from Table 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 Comparison

As was mentioned, two checkpoint protocols have been proposed previously for mobile environments. The protocol by Acharya and Badrinath [22] uses a two-phase rule to determine when processes need to save their state. The two-phase rule requires processes to create new checkpoints whenever they receive a message after having sent a message. Processes also have to create a checkpoint whenever the mobile host switches from foreign agents, and prior to disconnection. The protocol logs all messages that are exchanged between processes. Both the checkpoints and the messages are stored in the current foreign agent. As the mobile host roams between places, the checkpoints become scattered through the foreign agents.

Pradhan et al. [23] proposed two uncoordinated checkpoint protocols. The first protocol creates a checkpoint every time a process receives a message. The second approach creates checkpoints periodically, and logs all messages that are received. As in the protocol by Acharya and Badrinath, checkpoints and message logs are stored in the foreign agents. Pradhan et al. also propose three ways to deal with the problem of checkpoints becoming distributed through several nodes as the mobile host moves among cells.

Our protocol creates checkpoints whenever a local timer expires, and it only logs the unacknowledged messages at checkpoint time. The two previous protocols create checkpoints based on the messages exchanged by the processes. For certain patterns of communication they need to create a large number of checkpoints. The two previous protocols also need to log *all* messages, which can consume a large amount of disk. Our protocol uses soft and hard checkpoints to recover from different types of failures. The two previous protocols always assume hard failures, which means that they will always pay the extra cost of sending the checkpoints through the network. Our protocol does not rely on the foreign agents to store the checkpoints. In many cases, foreign agents will belong to some external organization that provides a mobile networking service. The protocol is not likely to be able to save the processes' states in these foreign agents. Having a computer in the home network that serves as stable storage also has two other advantages. First, the checkpoint protocol does not have to keep information about the location of the processes' states. Second, the computer can be made as reliable as the applications executed by the mobile hosts require (it is not possible to make *all* foreign agents highly reliable).

7 Conclusions

This paper describes a checkpoint protocol well adapted to the characteristics of mobile environments. The protocol is able to save consistent recoverable global states without needing to exchange messages. A process creates a new checkpoint whenever a local timer expires. 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 to be able to recover from soft failures.

Acknowledgments

We would like to thank the anonymous referees for their comments, and Jenny Applequist for her suggestions that helped to improve the readability of the paper.

References

[1] V. Bharghavan, "Challenges and solutions to adaptive computing and seamless mobility over heterogeneous wireless networks", *International Journal on Wireless Personal Communications*, 1996.

[2] D. Duchamp, S. Feiner, and G. Maguire Jr., "Software technology for wireless mobile computing", *IEEE Network Magazine*, pp. 2–18, November 1991.

[3] R. Katz, E. Brewer, E. Amir, H. Balakrishnan, A. Fox, S. Gribble, T. Hodes, D. Jiang, G. Nguyen, V. Padmanabhan, and M. Stemm, "The Bay Area Research Wireless Access Network (BARWAN)", in *Proc. of the Spring COMPCON Conference*, 1996.

[4] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: A highly available file system for a distributed workstation environment", *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, April 1990.

[5] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system", in *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995, pp. 172–183.

[6] M. Theimer, A. Demers, and B. Welch, "Operating system issues for PDAs", in *Proc. of the 4th Workshop on Workstation Operating Systems*, October 1993, pp. 2–8.

[7] M. Nemzow, *Implementing wireless networks*, McGraw-Hill Series on Computer Communications, 1995.

[8] N. Neves and W. K. Fuchs, "Using time to improve the performance of coordinated checkpointing", in *Proc. of the International Computer Performance & Dependability Symposium*, September 1996, pp. 282–291.

[9] J. S. Plank, *Efficient checkpointing on MIMD architectures*, Ph.D. thesis, Princeton University, June 1993.

[10] L. M. Silva and J. G. Silva, "Global checkpointing for distributed programs", in *Proc. of the 11th Symposium on Reliable Distributed Systems*, October 1992, pp. 155–162.

[11] Y. Tamir and C. H. Séquin, "Error recovery in multicomputers using global checkpoints", in *Proc. of the International Conference on Parallel Processing*, August 1984, pp. 32–41.

[12] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX", *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 1–24, February 1989.

[13] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging", in *Proc. of the 17th International Symposium on Fault-Tolerant Computing*, July 1987, pp. 14–19.

[14] N. Neves, M. Castro, and P. Guedes, "A checkpoint protocol for an entry consistent shared memory system", in *Proc. of the Thirteenth Annual Symposium on Principles of Distributed Systems*, August 1994, pp. 121–129.

[15] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems", *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204–226, August 1985.

[16] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing", *Journal of Algorithms*, vol. 11, no. 3, pp. 462–491, September 1990.

[17] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit", *IEEE Transactions on Computers*, pp. 526–531, May 1992.

[18] G. H. Forman and J. Zahorjan, "The challenges of mobile computing", *Computer*, pp. 38–47, April 1994.

[19] Y.-M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems", in *Proc. of the 11th Symposium on Reliable Distributed Systems*, October 1992, pp. 147–154.

[20] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations", in *Proc. of the 10th Symposium on Reliable Distributed Systems*, September 1991, pp. 12–20.

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

[22] A. Acharya and B. R. Badrinath, "Checkpointing distributed applications on mobile computers", in *Proc. of the Third International Conference on Parallel and Distributed Information Systems*, September 1994, pp. 73–80.

[23] D. K. Pradhan, P. Krishna, and N. H. Vaidya, "Recovery in mobile environments: Design and trade-off analysis", in *Proc. of the 26th International Symposium on Fault-Tolerant Computing*, June 1996, pp. 16–25.

[24] N. H. Vaidya, "A case for two-level distributed recovery schemes", in *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 64–73.

[25] C. Perkins, "IP mobility support", *Internet Engineering Task Force, Internet Draft (work in progress)*, February 1996.

[26] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, February 1985.

[27] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm", in *Proc. of the 4th Symposium on Reliable Distributed Systems*, October 1984, pp. 207–215.

[28] T. H. Lai and T. H. Yang, "On distributed snapshots", *Information Processing Letters*, vol. 25, pp. 153–158, May 1987.

[29] B. Randell, "System structure for software fault tolerance", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, June 1975.