Adaptive Recovery for Mobile Environments

A new checkpoint protocol that is well adapted to the unique challenges of mobile environments is described.

Nuno Neves and W. Kent Fuchs

MOBILE COMPUTING ENABLES USERS TO access and exchange information while they travel, roam in their home environments, or work at clients' sites. 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 [4, 6, 8].

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 acting 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 that of fixed hosts. These devices can execute general applications such as editors, spreadsheets, and databases. Portable computers 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 useful for disaster recovery after a fire, flood, or earthquake [6]. Currently several vendors are selling hardware support for wireless communication, using technologies such as 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 move between sites. Mobile hosts can also be lost or stolen. Transient failures due to power or connectivity problems can be frequent events.

In this article, 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 restarts its execution. Checkpoint protocols proposed in the past are not adequate for mobile environments because of disconnections (see "Related Work"). These protocols must either exchange messages during the creation of an application checkpoint [2, 9], or collect stored information during recovery [3, 5]. Another problem is that previous protocols 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 the available bandwidth with the application's messages, instead of using it to send the checkpoints to stable storage.

THIS ARTICLE PROPOSES A NEW COORDINATED checkpoint protocol for distributed systems. This protocol was designed to take into consideration the special characteristics of mobile environments. The protocol is able to store recoverable consistent 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 can be created in very little time. 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 is used to recover transient failures.

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 definitions. Otherwise, they will incur high overheads, or simply will not work correctly.

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.

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 to coordinate checkpoint creation or retrieve stored information during recovery 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.

Power is limited. The mobile host is often powered by batteries. Network transmissions and disk accesses are two of the most important sources of power consumption [4]. 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 the smallest possible number of accesses to disk.

Network characteristics are not constant. The various wireless technologies have completely different qualities of service [6]. For instance, a radio frequency LAN can have bandwidths between 2 and 20Mbps, but a wide-area LAN using cellular digital packet data may have a bandwidth of 19.2Kbps. Other different characteristics are cost, packet loss rates, and latency. The checkpoint protocol should adapt its behavior to the current network.

Different types of failures. Mobile host failures can be separated into two different categories. The first one includes all failures that cannot 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 failure 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.

Related Work

Uncoordinated checkpoint protocols save processes' states without having to exchange messages [3, 5, 12]. This is an interesting characteristic for mobile environments because checkpoints can continue to be stored while the hosts are disconnected. On the other hand, these protocols usually have to save a reasonable amount of information to guarantee deterministic reexecution. This can be a problem if the information has to be stored in the mobile host (there is typically a small amount of disk). Also, uncoordinated protocols usually need to exchange messages to garbage-collect the stored information [12]. During recovery, processes also send messages to find global states [5] or to obtain information stored by the other processes [3]. Most of the coordinated checkpoint protocols exchange messages while saving the application's checkpoints [2, 7, 9], which makes them unsuitable for mobile environments. Time-based coordi-

nated protocols do not need to send messages [7]. However, they rely on synchronized clocks or on tightly synchronized timers, which are difficult to guarantee with disconnections.

Two checkpoint protocols designed for mobile environments have been proposed [1, 10]. The protocol by Acharya and Badrinath [1] requires processes to create new checkpoints whenever they receive a message after sending a message. Processes also have to create a checkpoint whenever the mobile host switches from foreign agents (the hosts that forward the packets to the mobile host—see "Mobile Environment"), and prior to disconnection. The protocol logs all messages exchanged between processes. Both the checkpoints and the messages are stored in the current foreign agent. Pradhan et al. [10] 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 received. Checkpoints and message logs are stored in the foreign agents as in [1].

Our protocol creates checkpoints whenever a local timer expires, and it only logs the unacknowledged messages at checkpoint time. The two previous protocols might need to create a large number of checkpoints for certain patterns of communication. They also need to log all messages, which can consume a large amount of disk. Our protocol uses two types of checkpoints to recover from different types of failures. The two previous protocols always assume hard failures. 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 save the processes' states in these foreign agents.

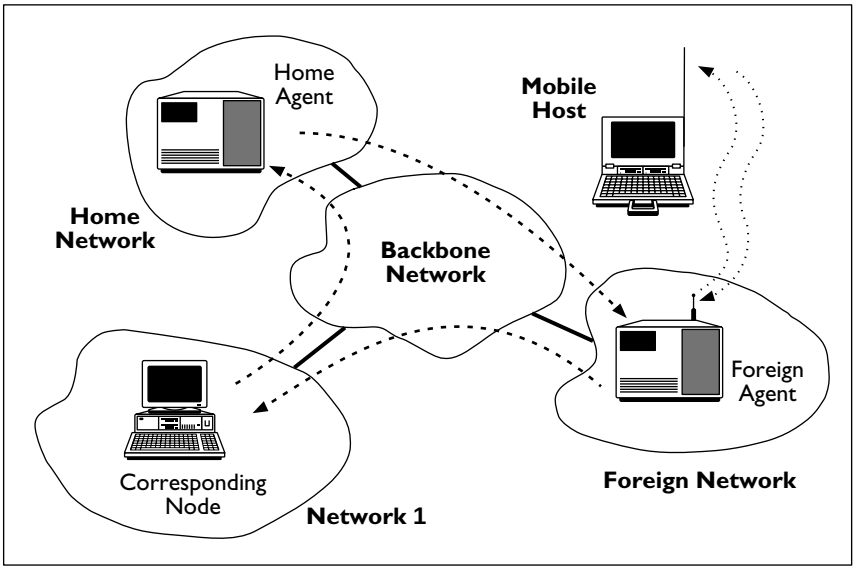

Figure 1. Mobile environment

Distributed System Model

Mobile Environment. The mobile environment model used in this article is based on the current Internet draft for mobile IP [8]. 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 *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 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.

Recoverable Consistent Checkpointing. A coordinated checkpoint protocol saves global states of an application that is executed by one or more processes. Processes run on

[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.

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 reexecute the application program and reread the logged messages. Recovery is only correct if the external results of the application reexecution are equivalent to one of the results of a failure-free execution [2].

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 reexecution 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.

```
// Pm            = Sender's identifier
// CNm           = Current checkpoint number of the sender
// timeToCkpm    = Time interval until next checkpoint
// mesgm         = Message contents
receiveMesg(Pm, CNm, timeTo Ckpm, mesgm):
      if ((CN = CNm) and (getTimeToCkp( ) > timeToCkpm))
         resetTimer(timeToCkpm);
      else if (CN < CNm) {    // orphan message
         createCkp( );
         resetTimer(timeToCkpm);
      }
      deliverMesgToApplication(mesgm);
```

Figure 2. Message reception

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 reexecution, but reduce the protocol's overheads.

The protocol creates two distinct types of process 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 the 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 correctly balancing soft and hard checkpoints, the protocol can keep its overheads approximately equal across 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 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.

Time-Based Checkpointing

The adaptive protocol uses time to avoid exchanging 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 at approximately 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 resynchronization mechanism to adjust timers during the application execution. 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 2). If the received interval is smaller, the process resets its timer with the received value. The resynchronization 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*, at each process to guarantee that the independently saved checkpoints verify the consistency property. 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 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 resends 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 *P*1 sends message *m*1. When *m*1 arrives, process *P*3 is still in its *CN*-1 checkpoint interval. To avoid a consistency problem, *P*3 first creates its *CN* checkpoint, and then delivers *m*1. *P*3 also resets the timer for the next checkpoint. Message *m*2 is an in-transit message that has not been acknowledged when process *P*2 saves its *CN* checkpoint. This message is logged in the checkpoint of *P*2. Message *m*3 is a normal message that indirectly resynchronizes the timer of process *P*2. It is possible to observe in the figure the effectiveness of the resynchronization mechanism.

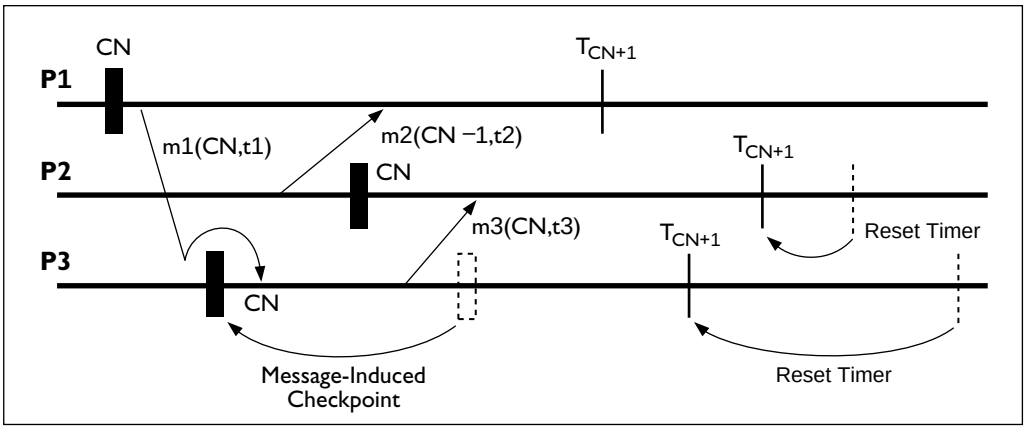

Figure 3. Time-based checkpointing

```
// 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

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 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 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 [11] 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 the worst 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, ..., and process $P2$ creates them whenever CN 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.

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

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 the process state locally, 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}.

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 cannot 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.

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 reconnection, the mobile host's process will be warned about the failure and will also have to roll back, 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, which allows independent recovery of soft failures, and minimizes the probability of global rollbacks due to failures of the mobile host.

When the mobile host reconnects, 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 *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.

Conclusion

This article has described 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, 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 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 article.

References

1. Acharya, A. and Badrinath, B.R. Checkpointing distributed applications on mobile computers. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Austin, Texas, Sept. 1994), pp. 73–80.
2. Chandy, K.M. and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 63–75.
3. Elnozahy, E.N. and Zwaenepoel, W. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Trans. Comput. 41*, 5 (May 1992), 526–531.
4. Forman, G.H. and Zahorjan, J. The challenges of mobile computing. *IEEE Comput. 27*, 4 (Apr. 1994), 38–47.
5. Johnson, D.B. and Zwaenepoel, W. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms 11*, 3 (Sept. 1990) 462–491.
6. Nemzow, M. Implementing wireless networks. *McGraw-Hill Series on Computer Communications*. McGraw-Hill, New York, 1995.
7. Neves, N. and Fuchs, W.K. Using time to improve the performance of coordinated checkpointing. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (Urbana, Illinois, Sept. 1996), pp. 282–291.
8. Perkins, C. IP mobility support. Internet Draft (work in progress), Internet Engineering Task Force, Feb. 1996.
9. Plank, J.S. Efficient checkpointing on MIMD architectures. Ph.D. dissertation, Princeton University, June 1993.
10. Pradhan, D.K., Krishna, P., and Vaidya, N.H. Recovery in mobile environments: Design and trade-off analysis. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, (Sendai, Japan, June 1996), IEEE, pp. 16–25.
11. Randell, B. System structure for software fault tolerance. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 220–232.
12. Wang, Y.-M. and Fuchs, W.K. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, (Houston, Texas, Oct. 1992), IEEE, pp. 147–154.

Nuno Neves *(nuno@crhc.uiuc.edu) is a Ph.D. candidate in computer science at the University of Illinois at Urbana-Champaign.*

W. Kent Fuchs *(fuchs@purdue.edu) is Head of the School of Electrical and Computer Engineering at Purdue University.*

This research was supported in part by the Office of Naval Research under contract N00014-95-1-1049. Nuno Neves was supported in part by the Grant BD-3314-94, sponsored by the program PRAXIS XXI, Portugal.

An earlier version of this article was presented at the High-Assurance Systems Engineering Workshop, October 1996. A more complete set of references can be found in the workshop presentation.