Recovery in the Mobile Wireless Environment Using Mobile Agents

Sashidhar Gadiraju and Vijay Kumar, *Member*, *IEEE*

Abstract—Application recovery in Mobile Database Systems (MDS) is more complex because of an unlimited geographical mobility of mobile units. The mobility of these units makes it tricky to store application log and access it for recovery. This paper presents an application log management scheme, which uses a mobile-agent-based framework to facilitate seamless logging of application activities for recovery from transaction or system failure. We compare the performance of our scheme with lazy, pessimistic, and frequency-based schemes through simulation and show that compared to these schemes, our scheme reduces overall recovery time by efficiently managing resources and handoffs.

Index Terms—MDS, mobile agents, PCS, coordinators, log unification, recovery.

1 Introduction

Wireless communication through PCS (Personal Communication Systems) or GSM (Global System for Mobile Communications) has become a norm of present day society. Cell phones are more common than watches and, in addition to being portable communication tools, they have become Web browsing platforms. Telecommunication companies are continuously improving the communication qualities, security, availability, and reliability of cell phones and trying to enhance its scope by adding data management capabilities, which is highly desirable. Motivated by such growing demand, we envision an information processing system based on PCS or GSM architecture, which we refer to as the Mobile Database System (MDS). It is essentially a distributed client/server system where clients can move around freely while performing their data processing activities in *connected*, *disconnected*, or *intermittent* connected mode. The MDS that we present here is a ubiquitous database system where, unlike conventional systems, the processing unit could also reach data location for processing. Thus, it can process debit/credit transactions, pay utility bills, make airline reservations, and other transactions without being subject to any geographical constraints. Since there is no MDS type of system available, it is difficult to identify the transaction volume at mobile units, however, the present information processing needs and trends in e-commerce indicate that transaction workload at each mobile unit could be high and MDS would be a useful resource to organizations and individuals alike.

Although MDS is a distributed system based on client server paradigm, it functions differently than conventional centralized or distributed systems and supports diverse applications and system functionalities. It achieves such diverse functionalities by imposing comparatively more constraints and demands on MDS infrastructure. To manage system-level functions, MDS may require different transaction management schemes (concurrency control, database recovery, query processing, etc.), different logging scheme, different caching schemes, etc. The topic of this paper is log management for application recovery through the use of mobile agents.

Application recovery [25], [26], unlike database recovery, enhances application availability by recovering the execution state of applications. For example, in MDS or in any distributed system, a number of activities related to transactions' execution, such as transaction arrival at a client or at a server, transaction fragmentation and their distribution to relevant nodes for execution, dispatch of updates made at clients to the server, migration of a mobile unit to another cell (handoff), etc., have to be logged for recovery. In recovery, the application recovery module recreates the execution state of application, which existed just prior to the failure and normal execution resumes.

Application recovery is relatively more complex than database recovery because

1. there are a large numbers of applications required to manage database processing,
2. presence of multiple application states, and
3. the absence of the notion of the "last consistent state."

This gets more complex in MDS because of

1. unique processing demands of mobile units,
2. the existence of random handoffs, and
3. the presence of operations in connected, disconnected, and intermittent connected modes.

Furthermore, it is not possible to store the entire log reliably at one location and retrieve it efficiently, which makes it very difficult to "see" the entire log for recovery. We argue that for MDS, the use of conventional approaches for managing log, even with modifications, would impose an unmanageable burden on the limited channel capacity and, therefore, reject their use.

• *The authors are with the School of Interdisciplinary Computing and Engineering, Computer Networking, University of Missouri-Kansas City, 5100 Rockhill Road, Kansas City, MO 64110. E-mail: sgadiraju@acm.org and kumarv@umkc.edu.*

Manuscript received 25 Nov. 2002; revised 2 June 2003; accepted 21 July 2003.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number 12-112002.

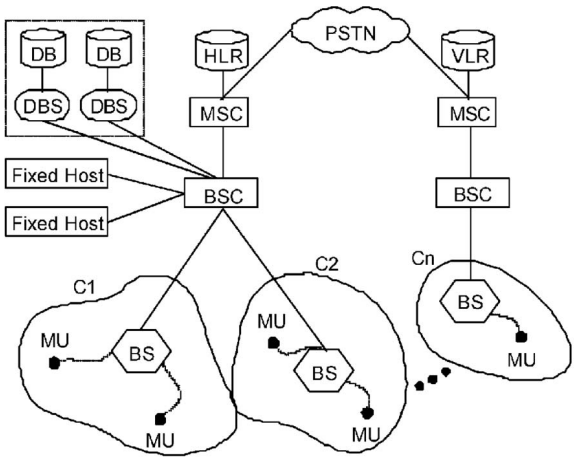


Fig. 1. A reference architecture of Mobile Database System (MDS).

In this paper, we take these challenges and present an efficient logging scheme, which stores, retrieves, and unify fragments of application log for recovery within the constraints of MDS. We recognize and exploit the unique processing capability of mobile agents for dealing with geographical mobility and use them to develop our log management scheme, which is scalable, that is, any new application can be added or existing ones can be deleted dynamically. We claim that our contribution helps to develop robust and highly available mobile information management systems, which are the backbone of e-commerce and m-commerce platforms. We are motivated by the work presented in [29], [30] to use mobile agents for system related problems. To the best of our knowledge, no previous work has exploited the unique processing capability of mobile agents in developing a log management scheme.

To show the superiority of our scheme, we compare its performance with lazy, pessimistic, and frequency-based schemes through simulation and report that, in all cases, our algorithm minimizes wireless cost and time. We begin with a reference architecture of MDS, which we have used for developing our logging scheme.

2 Reference Architecture of Mobile Database System and Transaction Execution

Fig. 1 illustrates our reference architecture of Mobile Database System (MDS). It is a distributed multidatabase client/server system based on PCS or GSM.[1] We have added a number of DBSs (database Servers) to incorporate data processing capability without affecting any aspect of the generic mobile network [3].

A set of general purpose computers (PCs, workstations, etc.) are interconnected through a high-speed wired network, which are categorized into Fixed Hosts (FH) and Base Stations (BS) or mobile support stations (MSS). One or more BSs are connected with a BS Controller or Cell Site Controller (BSC) [9], which coordinates the operation of BSs using its own stored software program when commanded by the MSC (Mobile Switching Center). We also incorporate some additional simple data processing capability in BSs to handle the coordination of transaction processing.[2]

Unrestricted mobility in PCS and GSM is supported by wireless link between BS and mobile units such as PDA (Personal Digital Assistants), laptop, cell phones, etc. We refer to these as Mobile Hosts (MH) or Mobile Units (MU) [9], [12], which communicate with BSs using wireless channels [9]. The power of a BS defines its communication region, which we refer to as a *cell*. The size of a *cell* depends upon the power of its BS and also restricted by the limited bandwidth of wireless communication channels. Thus, the number of BSs in MDS defines the number of cells. In reality, a high power BS is not used because of a number of factors [9], [12] rather a number of low power BSs are deployed for managing movement of MUs. A MU may be in a powered off or idle state (doze mode) or it may be actively processing data and can freely move from one cell to another. When a MU crosses a cell boundary, it is disconnected from its last BS and gets connected to the BS of the cell it enters. In such intercell movement, the *handoff* mechanism makes sure that the boundary crossing is seamless and data processing is not affected.

A DBS provides full database services and it communicates with MUs only through a BS. DBSs can either be installed at BSs or can be a part of FHs or can be independent to BS or FH. A MU is unable to provide reliable storage as provided by conventional clients and, for this reason, it usually relies on the static nodes (FH or BS) to save its data. This is especially true for activities such as recovery, logging, concurrency control, data caching, etc. It is possible to install DBS at BSs, however, we argue against this approach. Note that BS is a switch and it has specific tasks to perform, which does not include database functionality. To work as a database server, the entire architecture of a BS (hardware and software) may have to be revised, which would be unacceptable from a mobile communication viewpoint. We argue that mobile database functionality and wireless communication should be modular with minimum degree of overlap on their functionality. For these reasons and for the reason of scalability, we created DBSs as separate nodes on the wired network, which could be reached by any BS at anytime.

We describe transaction execution on MDS to introduce the problems of log management for application recovery. We use a mobile transaction model referred to as "*Mobilaction*," which we developed [20]. Here, we present only the main components of *Mobilaction* and details can be found in [20].

A *Mobilaction* (T_i) is defined as $T_i = \{e_1, e_2, \ldots, e_n\}$, where e_i is an "*execution fragment*." Each e_i represents a subset of the total T_i processing. A T_i is requested at a MU, it is fragmented [20], and are executed at the MU and at a

1. There are some difference in GSM and PCS architecture, however, these diffferences do not affect MDS functionality and we refer to both whenever necessary.

2. Note that, in GSM [12], BSs do have additional data processing capability so our assumption is in line with wireless system available functionality.

set of DBSs. We refer to the MU where a T_i originates or initiates as H-MU (Home MU) and the BS where H-MU initially registered as H-BS (Home BS). Note that no fragment of a Ti is sent to another MU for execution. This is because, in MDS, a MU is a personal unit and its use is controlled by its owner who can switch it off or disconnect it from the network at any time. This could force the Ti to fail unnecessary. Furthermore, other MUs may not have necessary data to process the fragment generated by another MU, in which case the fragment will end up at a DBS. Also, transfer of e_is to other MUs will incur wireless communication overhead which could be prohibitive.

In MDS, like conventional distributed database systems, a coordinator (CO) is required to manage the commit of T_i [20] and its role can be illustrated with the execution of a T_i. A T_i originates at H-MU and the H-BS is identified as the holder of the CO of T_i. H-MU fragments Ti extracts its e_i, sends $T_i - e_i$ to the CO and begins processing e_i. H-MU may move to other cells during the execution of e_i, which must be logged for recovery. At the end of the execution of e_i, H-MU updates its cache copy of the database, composes update shipment, and sends it to the CO. CO logs the updates from H-MU.

Upon receipt of $T_i - e_i$ from H-MU, the CO splits $T_i - e_i$ into e_j's $(i \neq j)$ and sends them to a set of relevant DBSs for execution. The H-MU may suffer a handoff and the CO may change. We explain later how a CO change is handled. Each DBS processes its fragment and updates its own database. Note that the presence of handoff may delay the execution and commit of a T_i. In this situation, even a *small* T_i may appear as a *long-running* T_i. Thus, the meaning of *long-running* T_i on MDS could be 1) a *small* T_i (such as debit/credit) may take a long time to run because of frequent handoffs and 2) the T_i does access a large number of data items, such as the preparation of bank customer monthly statements, and takes a long time to execute in the absence of any handoff. It is, however, meaningless to run statement preparation transactions on MU and long-running transaction in our case will be mostly of 1) type.

The size of information to be logged depends upon the number of handoffs a transaction suffers. Each event has to be recorded so that the information about the event and the location of the event are known for recovery. We elaborate this in later sections.

The selection of a component (BS, MU, DBS, and MSC) for housing CO module for a T_i is crucial because it affects its execution [20]. We argue that a MU is not a good choice for housing a CO because of the following limitations:

1. no direct communication with other processing nodes, especially with DBSs,
2. limited storage and reliability,
3. limited power supply,
4. unpredictable handoffs, and
5. limited availability.

However, in the case where T_i is entirely processed at the H-MU, it can act as a CO. A DBS is a database server and is not equipped with wireless communication capabilities, which is necessary for a CO. This leaves us to select either a BS or a MSC as a suitable candidate for housing COs. A MSC controls a large number of BSs and manages all handoffs. Adding coordinating responsibility will significantly increase its communication overhead. Furthermore, all BSs are well connected through the wired network, which is an important requirement of CO. For these reasons, we decided to select BS for coordinating transaction execution, but not for executing a T_i or an e_i because doing so would be undermining the task of DBSs and affecting the scalability of MDS. Note that the selection of BS or MSC for coordination may only affect the performance but not our scheme.

A MU may suffer handoffs during the processing of its e_is and T_i. During its movement, the MU will be disconnected and connected with different BSs one at a time, which will affect the connection of T_i with their CO too. In [20], we proposed handling this 1) statically or 2) dynamically. In a static approach, once a BS is selected as the location of T_i's CO, it remains so until T_i commits. In a dynamic method, the role of the CO changes with the movement of the MU (see [20] for further detail).

We decided to use static approach for the management of COs to minimize wireless communication overhead and to minimize the cost of control data dispatch to new COs. The problem with static approach is the increasing physical distance between the MU and the BS holding the COs of its T_i, which is likely to affect and hinder the communication between them. We have managed to alleviate this problem with our mobile agent-based framework, which we introduce in Section 6.

3 Recovery Problem Specification

MDS recovery process is significantly more complex than conventional systems mainly for the following reasons:

1. **MUs' stability**. The unlimited portability of MUs makes them vulnerable to all kinds of failure. For example, it may run out of its limited battery power, it may run out of its disk space, it may be affected by airport security, or user may physically drop the MU, and so on. Any of these events affect its functionality and a recovery algorithm must take these into consideration.
2. **Limited wireless bandwidth**. This severely affects its communication capability. During recovery, a MU may require communicating with BS or with other MUs, but it may not get any free channel for communication. This limitation could seriously affect recovery.
3. **Random Handoff**. MUs may be subjected to handoff randomly. A *handoff* may affect recovery mainly because the location of the desired MU may not be immediately available for communication.

3.1 Application Log Management

One of the most important activities of any recovery scheme is the management (creation, storing, and processing) of application log. An efficient recovery scheme requires that the log management must consume minimum system resources and recreate the execution environment as soon as possible after MU reboots. For application recovery, the H-MU and the server must

build a log of the events that change the execution states of T_i. Messages that change the log contents are called *write* events [15]. The exact *write* events depend on the application type. In general, the H-MU records events like

1. the arrival of a T_i,
2. the fragmentation of T_i,
3. the assignment of a CO to a T_i,
4. the mobility history of H-MU (handoffs, current status of the log, its storage location, etc.), and
5. dispatch of updates to the DBSs.

The DBSs may record similar events in addition to events relating to the commit of T_i.

In conventional distributed systems, log management is straightforward since no mobility is involved and a single stable storage area is available for storing log. In MDS, a MU cannot be relied upon and, therefore, it is necessary to store the log information at some stable place that can survive MU failure. Schemes that provide recovery in PCS failure use the BS where the MU currently resides for storing the log [15], [17]. Note that managing log for PCS failure is relatively easy because it does not support T_i processing.

Our objective is to utilize the unique processing capability of mobile agents in managing application log for efficient application recovery, which will conform to MDS limitations and mobile discipline constraints. We aim to achieve this conformity and desired efficiency by incorporating the following properties in our scheme:

1. communication overhead (wired/wireless) should below,
2. recovery time should be minimal, and
3. easy deployment of recovery schemes in the network.

Properties 1 and 2 are understood easily, but 3 needs some explanation, which we provide in Section 5.

4 Related Work

Works reported in [1], [7], [15], [17], [18], [21], [22], [28] deal with recovery related issues in mobile systems. To the best of our knowledge, none of these reports has investigated these problems for highly dynamic systems such as MDS and none has used mobile agent technology to develop solutions. They mainly deal with distributed applications (may or may not be database transactions) running on a number of mobile devices.

Work reported in [21] discusses mobile systems with low-bandwidth wireless communication as simple I/O devices or as full-fledged database servers. Global checkpoint-based schemes like [1] are not useful because they consider mainly applications running on multiple MUs unlike MDS where an e_i of a T_i runs only on H-MU and all other e_i's execute on different DBSs. We need schemes that perform efficient log management for a single MU. Hence, asynchronous recovery schemes presented in [15], [17], [18], [22] are better suited than schemes of [7], which require synchronization messages between participating processes.

Lazy and Pessimistic schemes (asynchronous schemes) are reported in [15]. In a lazy scheme, logs are stored in the BS and, if the MU moves to a new BS, a pointer to the old BS is stored in the new BS. The pointers can be used during failure to recover the log distributed over several BS. This scheme has the advantage that it incurs relatively less network overhead during handoff as no log information needs to be transferred. Unfortunately, this scheme has a large recovery time. In the pessimistic scheme, the entire log and checkpoint records, if any, are transferred at each handoff. Hence, the recovery is fast but each handoff requires large volumes of data transfer.

The work reported in [17] presents two schemes based on the MU's *movement* and uses independent checkpointing and pessimistic logging. In these schemes, the list of BSs where the log is distributed is transferred during a handoff. In the *distance-based* scheme, log unification is done when the distance covered by MU increases above a predefined value. In the *frequency-based* scheme, log unification is performed when the number of *handoffs* suffered by the MU increases above a predefined value. After unifying the log, the distance or handoff counter is reset. These schemes are a trade off between the lazy and the pessimistic strategies.

We believe that the existing mobile network framework, as suggested in the above papers, is not efficient for full-fledged database transactions (T_is) running at DBSs and MUs. In the above schemes, the location change of MU has to be updated by DBSs, which would be a big disadvantage. To overcome this, mobile IP was introduced. In [22], log recovery based on the mobile IP architecture is described where BSs store the actual log and checkpoint information and H-BS or the *home agent* as defined in [13] maintains the recovery information as MU traverses. This scheme has the advantage that log management is easy and the DB servers need not be concerned with the MU location update, but it suffers when the MU is far away from home. Consequently, recovery is likely to be slow if the home agent is far from the MU. The other problem with using mobile IP is *triangular routing* where all messages from the DB server to the MU have to be routed through the home agent. This invariably impedes application execution.

The schemes discussed so far do not consider the case where a MU recovers in a BS different than the one in which it crashed. In such a scenario, the new BS does not have the previous BS information in its VLR and it has to access the HLR to get this information [8], which is necessary to get the recovery log. HLR access may increase the recovery time significantly if it is stored far from the MU. A similar disadvantage can be observed in the mobile IP scheme of [22], where the MU needs to contact the home agent each time the MU needs recovery.

In the next section, we first present the mobile-agent paradigm and then a mobile agent-based architecture is described for logging. This agent-based framework provides a platform for implementing our scheme based on the distributed logging approach, which reduces recovery time while keeping the total network cost manageable.

5 A Mobile Agent-Based Log Management Scheme

A *mobile agent* is an autonomous program that can move from machine to machine in heterogeneous network under its own control [2]. It can suspend its execution at any point, transport itself to a new machine, and resume execution from the point it stopped execution. An agent carries both the code and the application state. Actually, the mobile agent paradigm is an extension of the client/server architecture with code mobility. Some of the advantages of mobile agents as described in [10], which we want to exploit are:

- **Protocol Encapsulation.** Mobile agents can incorporate their own protocols in their code instead of depending on the legacy code provided by the hosts.
- **Robustness and Fault Tolerance.** When failures are detected, host systems can easily dispatch agents to other hosts. This ability makes the agents fault-tolerant.
- **Asynchronous and Autonomous Execution.** Once the agents are dispatched from a host, they can make decisions independently and autonomously.

The last advantage stated above is particularly useful to the wireless environment where maintaining a connection throughout an executing T_i may not be economical or necessary. In such cases, the agents can visit the destination, perform any required processing, and bring the final data to the origin thereby removing the need for a continuous wireless connection. For example, an agent can take a *Mobilaction* from a MU, execute it at the most suitable node (could be remote), and bring the result to the MU.

Agents do have disadvantages and the one, which is likely to affect our scheme, is its high migration and machine load overhead [4]. This overhead must be minimized for improving the performance and, in our architecture, we achieve this by using agent services with "only when needed approach."

It is not possible to develop a scheme, which optimizes the performance at all levels and in all different situations. For this reason, some recovery schemes improve the performance by targeting to minimize the communication overhead, some might concentrate on total recovery time, some may optimize storage space, etc. Thus, each scheme involves certain trade offs.

When these issues are taken into consideration, it becomes necessary to build a framework that supports the implementation of the existing schemes and should also be able to support any new scheme. The framework should support the activation/deactivation of a scheme depending on the particular environment in which it offers best performance. Such a framework should abstract the core BS software (which handles the registration, handoff, etc. activities) from handling the recovery procedures, thus allowing for better recovery protocols to be implemented without the need for changing the core software. The framework may also support a rapid deployment of the recovery code without much human intervention. In our MDS, the CO module resides in the BS. It splits T_i's into e_i's if necessary, and sends some of them to a set of DBSs. This requirement asks for specific intelligence to be embedded in the BS code.

T_i's initiated by MU may use different kinds of commit protocols like 2-phase commit or 3-phase commit or TCOT (Transaction Commit On Timeout) [20]. The CO module needs to support all of these. If such a module at a BS does not support a particular protocol, then there should be an easy way to access such code. An extension to this is that, when a new efficient protocol is introduced, all BSs should be able to upgrade to this as easily as possible and with little or no human intervention.

From the perspective of MU log recovery, we need an architecture which supports intelligent logging and able to incorporate any future developments without any difficulty. Most papers suggest the BS as the stable storage for the MU logs. Some recovery schemes specify that the logs move along with the MU through a multitude of BSs. The new BS should be able to handle the logs in the same way as the previous one did or log inconsistency might result.

We argue that the flexibility and constraints mentioned above could be successfully incorporated on a mobile-agent-based architecture under which the code necessary for recovery and coordination can be embedded in the mobile agents. The CO can be modeled as a mobile agent and can be initiated by the MU itself if necessary. If during a handoff the new BS does not support a specific logging scheme, then the agent in the previous BS which supports this can clone itself and the new replica can migrate to the current BS without any manual intervention. The same technique can be used in quickly populating the BSs with any new protocols. The mobile agent with the new protocol embedded in it can be introduced in any BS and it can replicate and migrate to other BS.

We present an architecture where we use mobile agents to provide a platform for managing logging. The architecture supports the independent logging mechanisms. We assume that each BS supports the functionality of mobile agents. We describe our architecture in terms of various agents and their logical functions required in the framework, but we do not describe the implementation details:

- **Bootstrap agent (BsAg).** This agent addresses a BS failure. Any agent that wishes to recover should register with the bootstrap agent. The BS initiates the bootstrap agent. Once loaded, this agent starts all the agents that have registered with it. These agents are coded in such a way that when they are started, they have the capability to read the log information they have created and act accordingly. The need for such an agent may be obviated if the mobile agent provides an automatic revival of the agents with their state intact.
- **Base Agent (BaAg).** This agent decides which logging scheme to use in the current environment. Such functionality can be decided by its own intelligence or can be given as an input. For every MU, the BA creates an instance of an agent that handles the recovery of *Mobilactions* based on the relevant logging scheme.
- **Home Agent (HoAg).** This agent handles *Mobilactions* for each H-MU. It is responsible for maintaining log

and recovery information on behalf of H-MU. H-MU sends log events to this agent, which is responsible for storing them on the stable storage of the BS. The *HoAg* is a BS interface to the MU for *Mobilactions*.

- **Coordinator Agent (CoAg).** This is the coordinator agent residing at each BS.
- **Event Agent (EvAg).** In addition to the above framework, the BS provides mobile agents with an interface to the various events taking place like registration of a MU, failure of a MU, *handoff* of MU, etc. The need for such a support arises because we try to abstract away the core BS functions from application recovery support. When any MU suffers handoff, its *HoAg* should know about it so that it can perform the required operations. The *EvAg* is the interface for the BS to the agent framework for dissemination of such information.
- **Driver Agent (DrAg).** The migration of a mobile agent during a handoff, involves the movement of its code and the actual data as explained in the previous section. This might generate considerable overhead [4] even if the actual log transfer is not much. To manage this, we introduce *driver agents (DrAg)* and Section 5.2 describes its working.

5.1 Interaction of *CoAg* and *HoAg*

A MU sends *Mobilaction* to its *HoAg*, which forwards it to the corresponding *CoAg*. If the *CoAg* needs to contact the MU, it does so through the MU's corresponding *HoAg*. When *CoAg* sends a write event to the *HoAg*, it stores it in its local store before sending it to the MU. Similarly, if any events come to the MU through user input, MU sends the corresponding log messages to the *HoAg*.

5.2 Action of Agents when Handoff Occurs

The *HoAg* moves along with the MU to the new BS in a handoff. Based on schemes like lazy and frequency-based, the agent may or may not take the stored logs along with it to the new BS. Instead of the whole *HoAg* with all its intelligence for log unification, interaction with the *CoAg*, etc., when a handoff occurs, a *driver agent (DrAg)* is sent along with the necessary log information to the new BS. The *DrAg* has a very light code whose main function is to see whether the code for *HoAg* is present in the new BS. If so, it requests the resident *BaAg* in the new BS to create an instance of the *HoAg* for this MU. If any compatible code is not present, then the *DrAg* sends a request to the previous BS's *BaAg*, which clones the necessary *HoAg* and sends the copy to the new BS.

When MU moves out of a BS, its log information is not deleted automatically, but it is stored unless notified otherwise by the agent of the MU. This facilitates the unification of logs when logs are distributed over a set of BSs.

6 Forward Strategy

All schemes reviewed earlier have assumed instant recovery of the MU after a failure, but [8] acknowledges the possibility where the MU might crash in one BS and recover in another BS. We define the time interval between the MU failing and its subsequent rebooting as *Expected Failure Time (EFT)*. Our scheme concentrates on such scenarios where the EFT is not so trivial that the recovery occurs instantaneously.

BS detects the failure of a MU and agents do not play any part in such detection. For example, if the communication between two MUs breaks down because of the failure of one of the MUs, then the corresponding BS will immediately know about this event. Similarly, BS also knows which MU has executed power-down registration, which MU has undergone a handoff, and so on. A BS also continuously pages its MUs (Sprint PCS system pages it's MUs after every 10 to 15 minutes without generating any overhead [24]) to learn their status and a MU also continuously scans the air by using its antenna to detect the strongest signal.

If the MU suffers a handoff, then the communication with the last BS is not broken until the connection with the new BS is established (soft handoff). These features of PCS allow MDS to detect MU failure. Thus, while a MU is executing its e_i's, its status is continuously monitored by the BS and any change in MU's situation is immediately captured by the Event Agent interface. Since this detection is system dependent, *EFT* (*Expected Failure Time*) tends to be an approximate value. The detection can be passed on to the *HoAg* in many ways. The MDS can provide an interface, which would allow the agents to wait for an event. Another approach would be to provide an agent with a readable system variable which would be set on any such event. The agent will periodically poll the variable to check if it is set. Both approaches are possible and easy to implement in languages such as Java in which many agent systems like IBM's Aglets and General Magic's Odyssey have been developed [6].

Since handoff does not occur in the above case as pointed out in [8], the new BS does not know the location of the old BS. This situation leads to the new BS contacting the *Home Location Register (HLR)* for the previous BS. The schemes presented in [8], [15], [17], [18] build upon such support. This might be a hindrance to fast recovery if the HLR happens to be far from the querying BS. Actually, the *Visitor Location Register (VLR)* is first queried for the previous BS information (Fig. 1), which is stored in VLR if both BSs happen to fall under the control of the same VLR. If BSs are under different VLRs, then the HLR of the MU has to be queried. Such information is stored in the HLR when a MU first registers with a BS.

In the lazy scheme provided in [8], the BS starts building up the log immediately upon failure of MU. In the schemes presented in [17], the MU explicitly issues a *recover* call to BS and BS begins the log unification. This raises certain questions in the event of the MU crashing and recovering in different BSs. If the log is to be unified immediately upon a failure, then it might be necessary for the new BS to wait for the old BS to finish its unification and then present its log. If the failure time is large or the total log size is small, then unification will be over by the time the new BS queries the previous BS. In such a case, recovery can be fast.

In the case of a relatively small *EFT* (*Expected Failure Time*) or a large log size (to be unified), the new BS must wait first for the unification and then for the actual log transfer. This results in increased recovery time and network cost. In such

cases, it might be preferable for the log unification to be done in the new BS if the list of BSs where the log is distributed is known. Such list is transferred in schemes provided in [17] and not for those in [8]. In the approach where the log is unified after a recovery call, the recovery time might not be small enough if the log size to be unified is small. In this case, the unification has to begin after getting the list of BSs involved, from the previous BS. Also, if the MU has not migrated to a new BS before recovery, then log has to be unified, which is likely to increase the recovery time.

Here, we present two schemes that help in reducing the recovery time. Our scheme of log unification is based on the number of handoffs which have occurred since the last log unification or the start of the transaction, whichever is later. The log is unified periodically when the number of handoffs occurred crosses a predefined *handoff_threshold*.

When a handoff occurs, the "*Trace*" information is transferred from the old BS to the new BS. This *trace* information is an ordered list of elements giving information about the BSs involved in storing MU's log. Each array element consists of two values; 1) the identify of this BS (BS_ID) and 2) the size of the log stored at $BS\_ID_i(Log\_Size_i)$. When a handoff occurs, then BS_ID of the new BS and a Log_Size value of zero are added to the end of the *trace*. The Log_Size value is updated whenever MU presents BS with some log information. Some optional parameters can also be present in the trace information that we discuss in the next paragraph. Since the trace does not contain the actual log contents and is mostly an array of BSs' identities and log sizes, it does not present a significant overhead during the handoff.

The scheme also assumes the presence of *EFT (expected failure time)* value. This value can be stored as an environment attribute accessible to *HoAg* of the MU at BS. If such support cannot be given by the system, then *HoAg* can also estimate *EFT* from MU's activities. If the agent estimates the *EFT*, then this value is also stored in the *trace* information. When the system detects MU failure, it intimates the agent framework through the Event Agent interface. This agent notifies the appropriate *HoAg* that starts the *EFT clock*. This clock is stopped to get the *Recorded_EFT* value, when the *HoAg* receives a MU recovery call, which can come from the MU in the same BS or from a different BS in which the MU has recovered. In either case, the agent residing in BS where the *EFT clock* is started estimates *EFT* as the new

$$EFT = (\mathrm{K}_1 * Recorded\_EFT) + (\mathrm{K}_2 * EFT),$$

where $\mathrm{K}_1 + \mathrm{K}_2 = 1$.

The new *EFT* is a weighted sum of the previous *EFT* and the Recorded_*EFT*. K_1 indicates the reliance on the Recorded_*EFT* while K_2 indicates the reliance on the previously calculated *EFT*. The values of K_1 and K_2 are functions of the environment. In a network where the failure time is relatively stable, K_2 is given more weight and in a network where the failure time varies frequently, K_1 can be given more weight.

To improve storage utilization, unnecessary records from the log are deleted. This garbage collection is done upon log unification. When a MU log is unified at a BS, a *garbage_collect* message is sent to all the BSs hosting the MU logs as specified in the trace *BS_ID* list. The previous BSs purge these logs on receiving this message. The *BS_ID* and the *Log_Size* lists are erased from the trace information at the current BS to reflect the unification and a single entry is created in the trace with the current BS id and the unified log size. Note that garbage collection is not an integral part of our scheme, it is here to improve the performance.

6.1 Forward Log Unification Scheme

In this section, we explain when to start log unification upon a failure. We first define the actions in BS when a MU fails. Since the *trace* information contains the size of the log stored at different BSs, the *HoAg* can estimate the time for log unification based on the network link speed and the total log size. We call this the *Estimated Log Unification Time* (*ELUT*),[3] which can be measured as:

$$Max\{BS_i\_Log\_Size/Network\ link\ Speed + Propagation\ Delay\},$$

for all BSs in trace.

The exact characterization of the ELUT value depends on many other factors. i.e., whether the BSs are located in the same VLR area or different areas, queuing delay, etc. The *HoAg* should take into consideration as many parameters available from the system as possible to estimate the ELUT accurately.

Log unification is started if $(\delta * ELUT) <= EFT$ or else it is deferred until a *recovery* call is heard from the MU. The *Unification factor* "δ" describes what fraction of the log unification will be done by the time the failure time of the MU comes to an end. The default value can be kept as 1, which indicates that we start log unification only if it can be totally completed by the time the MU is expected to complete its reboot. If the MU reboots in a different BS while the log is being unified in the previous BS, it has to wait for the unification to complete.

Variations of this scheme are possible if the *HoAg* can estimate the effective handoff time. Based on this value, if there is still a long time for the next handoff, then the log unification can start immediately upon a failure, as it is more probable that the failed MU will recover in the BS where it failed rather than in any other BS.

In the event that log unification is not performed because $(\delta * ELUT) > EFT$, the *HoAg* waits for the MU to recover. If the recovery happens in the same BS, log unification starts, but if the MU reboots in a different BS, then the *HoAg* transfers the *trace* information and the log stored at this BS when requested. In this case, the new BS has to perform the log unification after getting the trace information from the previous BS. Note that this *trace* contains the newly calculated *EFT* value.

6.2 Forward Notification Scheme

This scheme addresses the issue of time spent in getting the previous BS information from the HLR. To overcome this,

3. The actual log unification time is difficult to calculate and that is the reason why we try to estimate it. The expression given is a simple way to estimate it without complicating the recovery.

we propose a scheme involving *forward notifications*. When a MU fails in a particular BS and if the actual failure time (total duration before MU is rebooted) is not too high, there is a high probability that the MU will recover in a BS that is in the same VLR or in a BS that is in adjacent VLRs. Our scheme is based on this premise, which is admissible because a VLR is generally assigned to a whole city covered by a number of BSs [12]. Thus, a VLR and its adjacent VLRs cover a large area and the situation where the MU reboots in a nonadjacent VLR does not occur frequently. If the MU happens to restart in a nonadjacent VLR, then it must have been extremely mobile and most of the recovery schemes are not designed for such an unrealistic situation. The other implication is that the MU had been in the failed state for a longer period and so it is likely that the CO could have decided to abort the *Mobilaction*.

We assume that each VLR stores MU's status information (*normal*, *failed*, and *forwarded*). A number of situations may arise and we explain the action of the system in each case.

Action of the system when a MU fails. When a MU fails, its corresponding *HoAg* informs the VLR about this failure. The VLR first changes the status of MU in its database from *normal* to *failed*. The VLR then issues a message containing its own identity (e.g., identity of the VLR that sends this message), the identity of the failed MU, and the identity of the BS in which the MU crashed to its adjacent VLRs that the MU has failed. The adjacent VLRs store these messages until explicit *denotify* messages are received. The MU is recorded in these adjacent VLRs with the status as *forwarded*. We describe the various scenarios that may arise when the MU reboots.

Case 1: The MU reboots in the same BS where it crashed. In this scenario, the *HoAg* informs the VLR that the MU has recovered. The VLR then issues a *denotify* message to all the adjacent VLRs indicating that the *forward notification information* is no longer valid. The status of the MU is changed back to *normal* from *failed*.

Case 2: The MU reboots in a different BS but in the same VLR. First, the MU registers with the BS and the registration message is logged on to the corresponding VLR. This VLR identifies the status of the MU as failed and then it proceeds as in Case 1 and sends *denotify* messages to the adjacent VLRs. The status of the MU is changed back to *normal* from *failed*. The new BS then proceeds to perform log unification from the previous BS.

Case 3: The MU reboots in a different BS and a different VLR. The MU requests for registration. The corresponding VLR identifies the MU as a forward notified MU and returns the identity of the previous BS and the identity of the VLR to the *HoAg* of the MU in the recovered BS. The BS then proceeds to perform log unification from the previous BS. Simultaneously, the new VLR sends a *recovered* message to the previous VLR regarding the recovered status of the MU and also sends a *registration* message to the HLR regarding the registration of the MU in the new location. The *status* of the MU is changed to *normal* from *forwarded* in the new VLR. Upon receiving the *recovered* message, the previous VLR sends a *denotify* message to all adjacent VLRs except the one in which the MU recovered and removes the registration of the MU from itself as well.

In the situation where the MU recovers in a nonadjacent VLR that has not received the forward notifications, the new BS has to get the previous BS information from the HLR and then send the previous VLR a *recovered* message. Upon receiving this message, the previous VLR acts similar to the previous VLR of Case 3.

The forward notification scheme is unsuitable if the MU suffers failures with a very small *EFT*. In that case, the MU recovers in the same BS where it failed. Hence, the forward notifications and subsequent denotifications generate communication overhead. To alleviate this, we might delay the sending of these notifications immediately on failure of the MU. The *HoAg* waits for an *initial buffer time* before it notifies the VLR regarding the failed status of the MU. This time can be estimated by the *HoAg* in a way similar to the estimation of *ELUT* without compromising the performance.

We can reduce the overhead further if we can reduce the number of recipients of the notifications. If notifications are sent to only those VLRs, the BSs of which are nearer to the BS in which the MU fails, the communication overhead can be reduced.

7 Performance Study

We compare the performance of our scheme with *lazy* and *pessimistic* [15] and the frequency-based *movement* scheme [17]. The distance-based movement scheme [17] is quite similar to the frequency-based scheme and, hence, we do not include it in our simulations. In this section, we first present the simulation model and later discuss the performance results.

7.1 Simulation Model

We have assumed an MDS structure with 6×6 BSs (Base Station) arranged in a grid fashion with each cross point in the grid representing a BS. According to this model, all BSs have equal area and each BS has at most eight adjacent neighbors. Initially, 100 MUs (Mobile Unit) are randomly distributed among these 36 BSs so that, on the average, these BSs have been combined into groups of nine (3×3 grid) where each group represents a MSC (Mobile Switching Center) and a VLR attached to it. This presents us with four MSCs of equal size, each with nine BSs. Our objective in such grouping is to show the effect of forward notifications on the migration of MUs between BSs of different MSCs. The grouping of nine allows the MSC configuration of eight border BSs (which have borders with BSs from a different MSC) and one internal BS. In reality, there are many internal BSs, but we provide for the minimal configuration. The grouping of four (2×2 grid) does not provide for any internal BS and, hence, we do not consider it.

Each MU suffers handoff and failure according to a *Poisson* process with rates of λ_h and λ_f, respectively. The failure time at each failure is exponentially distributed with a mean of $1/\lambda_{EFT}$. Each MU sends log data to the corresponding registered BS with a time interval of mean $1/\lambda_c$, which is exponentially distributed. The log size transferred at each log event is constant and is equal to C_L. At any instant, the MU is involved in a single *Mobilaction*, with each *Mobilaction* containing C_T log events to be stored in the BS stable storage. Checkpoints are not

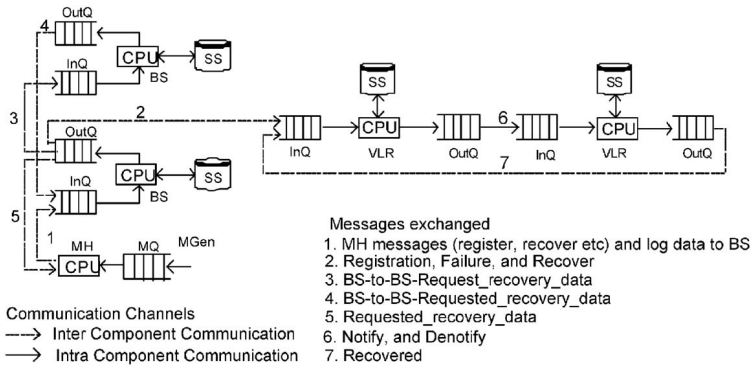


Fig. 2. Simulation model.

generated in the simulation and only message logs are used. When a *Mobilaction* ends, the corresponding logs are deleted from the stable storage. The cost of transferring a single log event on the wireless channel is $\alpha * C_L$, where α is the wireless link cost factor. Similarly, we define wired link cost factors β_1 and β_2, where β_1 is the cost factor on wired links for inter-MSC message transfer while β_2 is the cost factor for intra-MSC message transfers. We differentiate between β_1 and β_2 because inter-MSC BSs message transfer has to traverse the additional PSTN link (see Fig. 1), which is not required in an intra-MSC BSs message transfer.

The messages exchanges among relevant components of MDS are shown in Fig. 2. A MU consists of a *Mobilaction* generator shown as *MGen*, which generates log events. These events are sent to FIFO queue *MQ*. Transmission of an event from MQ does not start until its previous event is completely transferred from MQ. The BS receives the events from MQ through an input queue *InQ* and stores the events in the stable storage (SS). These events are deleted from the storage when garbage-collection requests are issued by other BSs. Such requests from other BSs are not stored in the SS. When BS needs to communicate with other BSs or MUs, the message is sent to the *OutQ* and the log events are sent to the stable storage to be stored. The *InQ* and *OutQ* are operated in a round-robin manner.

We study the performance of the various schemes in terms of the following costs:

- C_H: Handoff log management cost. It is the sum of message transfer cost (actual log and/or *trace* information) between the BSs during a single handoff and the resulting control messages (acknowledgements). For the lazy scheme, this cost is just the cost of creating the link to the previous BS. No log information is transferred. For the pessimistic scheme, this cost includes the cost of transferring the total log, and the cost of acknowledgement. For the movement and forward schemes, this cost includes the cost of *Trace* information, possibly log transfer cost (including log request cost), and acknowledgement cost.
- C_R: It is the cost for log retrieval or log unification cost incurred by the BS where a failed MU reboots. $\mathrm{C_R}$ is a measure of the recovery cost and can be represented as

$$C_R = Cost\ for\ log\ requests + Cost\ for\ log\ transfers + Cost\ for\ log\ unification\ waiting.$$

Note that C_R includes the transfer cost of control messages issued by the BS needed to request the log from other BSs where the log is distributed and the actual log transfer cost from those BSs. If a MU reboots in $\mathrm{BS_2}$ after a failure, but its log unification is still continuing at $\mathrm{BS_1}$ (its previous base station), then C_R includes the additional cost of waiting for the log unification to completed at $\mathrm{BS_1}$. Such a scenario can occur if immediately after a handoff, the MU fails while the log is being unified because the number of handoffs crosses the *handoff_threshold* (observe that this threshold can be taken as 0 in the pessimistic scheme, and as 8 in the lazy scheme). This scenario also occurs if a MU suffers a second failure and moves to a different BS while recovering from a previous failure. C_R does not include the cost for the subsequent transmission of the unified log to the MU over the wireless channel, as this cost is the same for different schemes.

- C_F: Failure Cost. It is the sum of costs included from the point a MU completely recovers from a failure to the next failure point when the BS is ready to transmit the unified log to MU. In other words, it is the total cost of recovering from a single failure. It includes any handoff costs that may occur, cost for forward notifications if any, the forward log unification cost (which is zero for the *movement* scheme), and the subsequent cost of sending the log to the BS where the MU reboots.

Table 1 lists the input parameter values that drive the simulator. We obtained the values of these parameters from sources like [15], [17], [20] and decided the values of the failure rate and the log transfer time for a log size of 1 over the wired and wireless channels. The simulations for movement and forward schemes are performed with the *handoff_threshold* of five. We set the failure time as 10, which is the time required for sending a single log message over the wireless channel from the MU to its *HoAg* in the BS. The upper range of handoff rate is set at 0.2 as any larger value results in excessive handoff drops as the failure time has a mean of 10. Even then, some handoffs are completely processed as the handoff rate follows a Poisson process and may some times be greater than its mean value. But, such

TABLE 1
Simulation Input Parameters

Input Parameters	Values
Failure Rate, λ_f	0.01
Failure Time, $1/\lambda_{EFT}$	10
Log sending rate, λ_c	0.1
Log size, C_L	1
Log events per *Mobileaction*, C_T	20
Wireless link cost factor, α	10
Wired link cost factor for Intra-MSC BSs message transfer, β_1	2
Wired link cost factor for Inter-MSC BSs message transfer, β_2	3
Unification factor, δ	1
Range of Handoff Rate	0.003125 0.2
EFT Parameters K_1, K_2	0.2, 0.8

handoffs are very few for the results to be meaningful. The handoffs are also dropped as it would not be possible to completely transfer the logs during the handoff in schemes like pessimistic. The *EFT* (Expected Failure Time) parameters are set so as to give more importance to the previous *EFT* value.

7.2 Simulation Results and Discussion

7.2.1 Effect of Handoff Rate λ_h on Handoff Cost C_H

Fig. 3 shows the relationship between handoff rate and handoff cost. The C_H increases with the handoff rate. This can be attributed to the fact that a *Mobilaction* log may be distributed in multiple intra-MSC BSs. The handoff cost (C_H) is the lowest for the *Lazy* scheme because no log or significant trace information is carried during a handoff and, hence, its cost does not vary with a change in λ_h. The *pessimistic* scheme fares the worst as it carries the whole log during each handoff. Both the *movement* scheme and *forward* scheme have nearly the same handoff cost. The *forward* scheme incurs marginally more cost than the *movement* scheme because of the additional information of the log sizes in the trace information. The difference in the costs between the two schemes is too small to observe at lower handoff rates, but it is visible at high λ_h when the number of BSs traversed and recorded in the *trace* information is more.

7.2.2 Effect of Handoff Rate, λ_h, on Log Retrieval Cost C_R

Fig. 4 shows the relationship between handoff rate and handoff cost. The *Lazy* scheme has the worst performance for C_R and is affected significantly with an increase in the handoff rate. This is because the number of BSs over which the log is spread increases rapidly with handoff rate resulting in increased recovery cost. Moreover, log recovery is sequential as there is no BS list in *lazy* scheme. Every BS has to contact the previous BS for log. The *pessimistic* scheme has the best performance because the log is already transferred during the handoff, but the cost increases with λ_h. This is because the probability of MU crashing in one BS

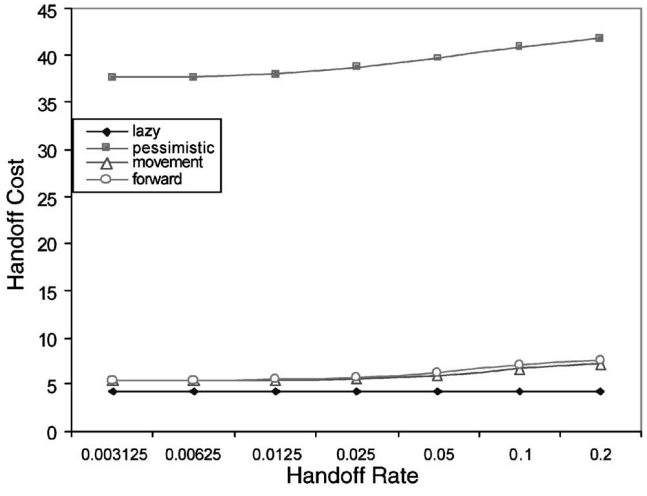


Fig. 3. Handoff cost with handoff rate.

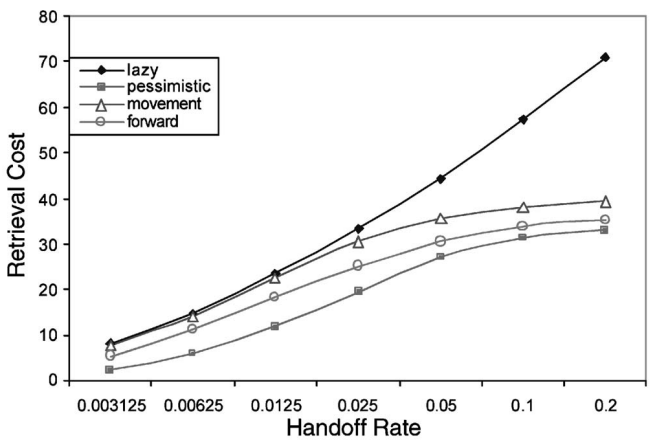


Fig. 4. Cost of log retrieval with handoff.

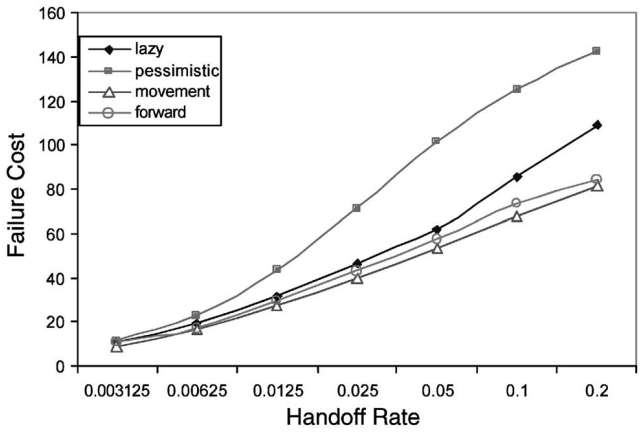


Fig. 5. Cost of failure with handoff.

and comes up in another BS increases where the log has to be transferred from the previous BS. The *movement* scheme fares better than the Lazy scheme due to the periodic log unification. The *forward* scheme performs better than *movement* because at lower λ_h, the forward unification helps in reducing the recovery cost. The effect of forward notification is not much as it is more likely that the MU comes up in the same BS as it has failed. But, at higher handoff rates, the forward notification also helps in reducing the recovery time.

7.2.3 Effect of Handoff Rate λ_h on the Failure Cost C_F

Fig. 5 shows the relationship between handoff rate and failure cost. We observe that the failure cost for a single failure is largest for the *pessimistic* scheme. This is because of complete log transfer at each handoff. As the handoff rate increases, so does the failure cost because the number of handoffs per failure increase. The *lazy* scheme performs better than the *pessimistic* scheme because log unification happens only on failure. However, as λ_h increases, C_F value increases rapidly because the cost for log unification rises with the number of BSs involved in storing the log. The *movement* scheme performs best among all the schemes because of its periodic log unification, which reduces the recovery time without increasing the handoff costs significantly. The *forward* scheme has a slightly greater cost than the movement cost because of the forward notifications and the subsequent denotifications. The forward unification contributes to the increase in the failure cost, but it has the advantage of reducing the recovery time.

8 Conclusion

In this paper, we presented a mobile agent-based framework for supporting application recovery in a mobile, wireless environment. We presented the *forward* strategy for improving the recovery time in situations where the failure time is nontrivial. Our framework is not restricted to the forward scheme, but it can support previous independent logging schemes. The simulation results show that the forward scheme improves the recovery time with a fairly consistent behavior in all the parameters simulated.

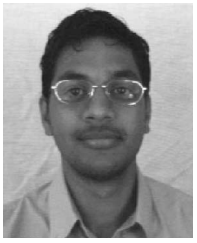
Acknowledgments

This research is supported by the US National Science Foundation under grant no. IIS 19979453.

References

[1] A. Acharya and B.R. Badrinath, "Checkpointing Distributed Applications on Mobile Computers," *Proc. Third Int'l Conf. Parallel and Distributed Information Systems,* pp. 73-80, 1994.
[2] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko, "AgentTCL: Targeting the Needs of Mobile Computers," *IEEE Internet Computing,* vol. 1, no. 4, 1997.
[3] M.H. Dunham and A. Helal, "Mobile Computing and Databases: Anything New?" *SIGMOD Record,* vol. 24, no. 4, pp. 1-9, Dec. 1995.
[4] G. Eleftheriou and A. Galis, "Mobile Intelligent Agents for Network Management Systems," *Proc. London Comm. Symp.,* 2000.
[5] T. Imielinski and B.R. Badrinath, "Mobile Wireless Computing: Solutions and Challenges in Data Management," *Comm. ACM,* pp. 19-27, Oct. 1994.
[6] J. Kiniry and D. Zimmerman, "A Hands-On Look at Java Mobile Agents," *IEEE Internet Computing,* vol. 1, no. 4, pp. 21-30, 1997.
[7] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. Software Eng.,* vol. 13, no. 1, pp. 23-31, 1987.
[8] P. Krishna, N.H. Vaidya, and D.K. Pradhan, "Recovery in Distributed Mobile Environments," *Proc. IEEE Workshop Advances in Parallel and Distributed Systems,* Oct. 1993.
[9] R. Kurupppillai, M. Dontamsetti, and F.J. Cosentino, *Wireless PCS.* McGraw-Hill, 1997.
[10] D.B. Lange and M. Oshima, "Seven Good Reasons for Mobile Agents," *Comm. ACM,* vol. 42, no. 3, 1999.
[11] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal and Optimal," *IEEE Trans. Software Eng.,* vol. 24, no. 2, pp. 149-159, Feb. 1998.
[12] M. Mouly and M.-B. Pautet, *The GSM System for Mobile Communications.* pp. 100-102, Cell & Sys Publications, 1992.
[13] C. Perkins, "Mobile Networking through Mobile IP," *IEEE Internet Computing,* pp. 58-69, Jan. 1998.
[14] P.A. Bernstein, "Concurrency Control and Recovery in Database Systems," pp. 226-236, 2003, www.research.microsoft.com/pubs/ccontrol.
[15] D.K. Pradhan, P. Krishna, and N.H. Vaidya, "Recovery in Mobile Environments: Design and Trade-Off Analysis," *Proc. 26th Int'l Symp. Fault-Tolerant Computing (FTCS-26),* June 1996.
[16] F.D. Schlichting and F.D. Schneider, "Failstop Processors: An Approach to Designing Fault-Tolerant Distributed Computing Systems," *ACM Trans. Computer Systems,* vol. 1, no. 3, pp. 222-238, 1983.
[17] T. Park, N. Woo, and H.Y. Yeom, "An Efficient Recovery Scheme for Mobile Computing Environments," *Proc. Eighth Int'l Conf. Parallel and Distributed Systems,* 2001.
[18] T. Park and H.Y. Yeom, "An Asynchronous Recovery Scheme Based on Optimistic Message Logging for Mobile Computing Systems," *Proc. 20th Int'l Conf. Distributed Computing Systems,* pp. 436-443, Apr. 2000.
[19] V. Kumar and M.H. Dunham, "Defining Location Data Dependency, Transaction Mobility and Commitment," TR 98-cse-1, Southern Methodist Univ., Feb. 1998.
[20] V. Kumar, M.H. Dunham, N. Prabhu, and A.Y. Seydim, "TCOT–A Timeout Based Mobile Transaction Commitment Protocol," *IEEE Trans. Computers,* vol. 51, no. 10, pp. 1212-1218, Oct. 2002.
[21] V.R. Narasayya, "Distributed Transactions in a Mobile Computing System," *Proc. IEEE Workshop Mobile Computing Systems and Application,* June 1994.
[22] B.. Yao, K.F. Ssu, and W.K. Fuchs, "Message Logging in Mobile Computing," *Proc. IEEE Fault-Tolerant Computing Symp.* pp. 294-301, June 1999.
[23] Y.-B. Lin and I. Chlamtac, *Wireless and Mobile Network Architectures.* Wiley Computer Publishing, 2001.
[24] Personal communication and Sprint PCS communication document, 2001.
[25] R. Barga and D. Lomet, "Phoenix: Making Applications Robust (demo paper)," *Proc. ACM SIGMOD Conf.,* June 1999.
[26] D. Lomet and G. Weikum, "Efficient Transparent Application Recovery In Client-Server Information Systems," *Proc. ACM SIGMOD,* June 1998.

[27] D.B. Lomet, "Application Recovery: Advances Toward an Elusive Goal," *Proc. Workshop High Performance Transaction Systems (HPTS '97)*, Sept. 1997.
[28] C.P. Martin and K. Ramamritham, "Recovery Guarentees in Mobile Systems," *Proc. Int'l Workshop Data Eng. for Wireless and Mobile Access*, pp. 1-7, Aug. 1999.
[29] G. Samaras, K. Karenos, P.K. Chrysanthis, and E. Pitoura, "VISMA Extendible Mobile-Agent Based Services for the Materialization and Maintenance of Personalized and Sharable Web Views," *Proc. 14th Int'l Workshop Database and Expert Systems Applications (Dexa '03)*, pp. 974-979, 2003.
[30] S. Weissman Lanzac and P.K. Chrysanthis, "Personalized Information Gatherinf for Mobile Database Clients," *Proc. Ann. Symp. Applied Computing*, pp. 49-56, Mar. 2002.



Sashidhar Gadiraju received the BE degree from the Osmania University, Hyderabad, India, and the MS degree from the University of Missouri, Kansas City, both in computer science. He worked as a programmer at Children's Mercy Hospital and Clinics from 2002 to 2003. His research interests include mobile computing, database systems, computer networking, and bioinformatics.

Vijay Kumar is a professor of computer science at the University of Missouri at Kansas City. His research areas are mobile computing, sensor technology, data warehousing, workflow, Web, and computational biology. He has published papers in the *ACM Transactions on Database Systems*, the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Mobile Computing*, *Information Systems*, *Data and Knowledge Engineering*, etc., and conferences such as ICDE, COMAD, COMPSAC, etc. He has served as program chair, conference chair, and as a program committee member on a number of national and international conferences and workshops. He has written three books on database systems. He is a member of the IEEE and the ACM.