On Tolerating Failures of Mobile Hosts and Mobile Support Stations

JinHo Ahn

Dept. of Computer Science, Kyonggi University, Suwon Gyeonggi-do, Republic of Korea

Summary
In this paper, we present two fault-tolerant protocols for mobile computing systems; a causal message logging protocol and a receiver-based pessimistic message logging protocol for tolerating failures of mobile hosts (MHs) and mobile support stations (MSSs) respectively. The systems raise several constraints such as limited life of battery power, mobility and disconnection of hosts and lack of stable storage. Existing causal message logging protocols in distributed systems can efficiently handle some among the constraints. However, they are unable to handle the other constraints such as mobility and disconnection of MHs, and lack of stable storage. Our causal logging protocol can handle all the constraints efficiently and improve asynchrony during recovery. Moreover, it maintains only one checkpoint for each MH. Our receiver-based pessimistic logging protocol reduces the failure-free overhead and improves scalability to a large number of MHs managed by each MSS compared with existing replication-based protocols.

Key words:
Distributed and mobile computing system, Fault-tolerance, Asynchronous checkpointing, Message logging, Recovery

1. Introduction

A mobile computing system extends a distributed computing system to support mobility of hosts by providing MHs with continuous network connections. In other words, it consists of MHs, MSSs that are fixed hosts and act as access points for MHs by wireless links, and other fixed hosts [1]. MHs have the following new features [1, 12, 21] that make them different from fixed hosts.

• Changes in MHs' locations: When an MH *h1* sends messages to an MH *h2*, the messages may have to be rerouted if *h2* moves from one cell to another. Searching an MH generally increases the delay and message complexity.

• Limited battery life each MH has: Each MH is occasionally powered by batteries. To minimize power consumption, it can power down important sources of power consumption such as network transmissions and disk accesses.

• Disconnection: A disconnected MH is unreachable from the rest of the network. While disconnected, the MH can not send or receive any message to or from other hosts.

• Vulnerability of MHs to failures: MHs can fail more often and concurrently than fixed hosts because the battery is discharged and contents in memory are lost, or the operating system crashes. Thus the MH's disk is not stable.

As mobile computing systems scale up, their failure rate may also be higher. Thus, they require the techniques to support fault-tolerance for MHs and MSSs respectively. Log-based rollback recovery is among the techniques, and requires that each process periodically saves its local state and logs the messages it received after having saved the state. Message logging protocols are classified into pessimistic [10, 20], optimistic [7, 11, 18], and causal [2, 3, 4, 6]. In this paper, we present a causal message logging protocol and a receiver-based pessimistic message logging protocol for tolerating failures of MHs and MSSs respectively. Although pessimistic logging protocols provide fast recovery, they require a process to save synchronously every received message on stable storage before it delivers the message to the application. Thus, they may be unsuitable for minimizing power consumption of MHs during failure-free execution. Although optimistic logging protocols result in low failure-free overhead compared with pessimistic logging ones, they may cause live processes to rollback to their previous globally consistent set of states because failed processes lose their volatile logs. Thus, they may be unprofitable for minimizing power consumption of MHs during recovery. Causal logging protocols used in distributed systems require each process to save received messages in its volatile memory during failure-free execution and restrict the rollback of any failed process to the most recent checkpoint on stable storage even in concurrent failures. Their features may be profitable for

Manuscript received June 5, 2007
Manuscript revised June 20, 2007

minimizing power consumption of MHs during failure-free execution and recovery. However, they require the mechanisms for handling the constraints such as mobility and disconnection of MHs, and lack of stable storage. Furthermore, these constraints make other recovery algorithms, based on the causal logging protocols in distributed systems, unsuitable for mobile computing. For examples, some recovery algorithms [2, 3, 6] prevent live processes from continuing executing during recovery and require some synchronous logging to stable storage while recovery is ongoing. Although Elnozahy's recovery algorithm [4] solves the problems, it requires a central recovery leader, which may be a performance bottleneck. Moreover, it results in nontrivial election overhead and if the leader fails continuously before it completes its recovery procedure, the other recovering processes should continue being blocked. Additionally, if it were integrated with asynchronous checkpointing, it could result in inconsistency problems in case of concurrent failures. Our causal message logging protocol can handle all the constraints of MHs efficiently and solve all the stated problems of Elnozahy's recovery algorithm because each recovering process is responsible for only its recovery.

As mobile computing is increasingly gaining popularity, MSSs should manage a large number of mobile hosts. Thus, they may be single points of failure and potential performance bottlenecks. Especially if a home agent in Mobile IP [16] fails, all the MHs managed by it can not communicate with other hosts. The problems can be solved allowing multiple MSSs to be assigned to a network. Existing fault-tolerant protocols to mask failures of MSSs have used replication techniques [8]. In the protocols, all multiple MSSs on a network must always maintain the location information about every MH registering with the network respectively. Thus, the protocols result in high synchronization overhead among replicas and are not scalable. Our receiver-based pessimistic logging protocol reduces the failure-free overhead and improves scalability to a large number of MHs managed by each MSS compared with the replication-based protocols.

The rest of the paper is structured as follows. In section 2, we explain the distributed and mobile computing system model assumed in this paper. Sections 3 and 4 propose efficient protocols for tolerating failures of MHs and MSSs respectively, and then section 5 compares our work with related works. Finally, we conclude this paper in section 6.

2. System Model

We assume the mobile computing system like in figure 1. It consists of a set of MHs and a set of fixed hosts and is asynchronous: there is no global clock and no limits of the relative speeds of processors. A fixed host may be a regular host (FH) or a MSS. A MSS is connected by a fixed wired network, which provides reliable FIFO delivery of messages. A cell is a geographical area in which a MSS supports MHs. Each MH can directly communicate with a MSS by a reliable FIFO wireless network only if the MH is in the cell covered by the MSS. Hosts in the system communicate with each other only through messages. To manage a large number of MHs in a network, multiple MSSs, not one may be assigned to the network like in figure 1 [8]. The execution of each process on each host is piecewise deterministic [5]. We assume that hosts fail, in which case they lose contents in their volatile memories and stop their executions, according to the fail stop model [5]. Events of processes in a failure-free execution are ordered using Lamport's happened before relation [13].

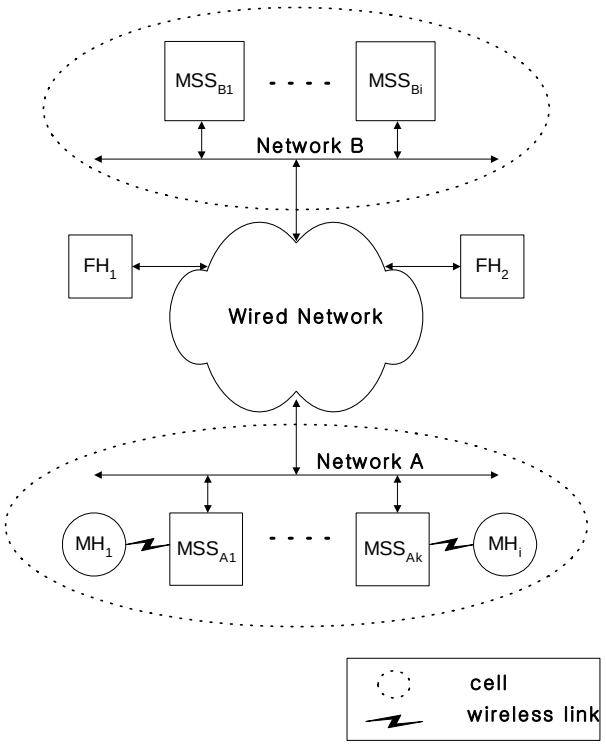

Fig. 1. Mobile Computing System Model

3. The Protocol for tolerating failures of MHs

3.1 Traditional causal message logging protocols

Traditional causal logging protocols used in distributed systems have the advantages inherited from pessimistic

and optimistic logging protocols [3]. First, each process piggybacks determinants in its volatile memory on every message to be sent to each other in order to prevent orphan processes like in pessimistic logging protocols. The determinant of a message includes the identifiers of the sender process (SID) and the receiver process (RID), the send sequence number assigned to the message by the sender (SSN), and the receive sequence number assigned to it by the receiver (RSN). Therefore this advantage is profitable for minimizing the number of local checkpoints maintained for each MH's recovery because the protocols may restrict the rollback of any failed process to the most recent checkpoint on stable storage even in concurrent failures. Second, each process logs all determinants piggybacked on each received message in its volatile memory like in optimistic logging protocols. This advantage is profitable for minimizing the number of network transmissions of MHs during failure-free execution. However, they can not handle mobility and disconnection of MHs, and lack of stable storage.

Some recovery algorithms [2, 3, 6] of the protocols prevent live processes from continuing executing during recovery, which may reduce the speed of the computation of the entire system. Elnozahy's recovery algorithm [4] solves the problem. However, it requires a central recovery leader, which may be a performance bottleneck. Moreover, it results in nontrivial election overhead and if the leader fails continuously before it completes its recovery procedure, the other recovering processes should continue being blocked. Additionally, if it were integrated with asynchronous checkpointing, it could results in inconsistency problems in case of concurrent failures like in figure 2. For example, we assume that p2 sends m_3 to p3 and fails after it has received m_1 and m_2 from p1 and p3 respectively. In Elnozahy's algorithm, p2 restores its latest checkpoint and then sends p1 and p3 recovery messages to know whether p1 and p3 are recovering or live processes. If p3 receives the recovery message from p2 after p3 has received m_3 from p2, taken its local checkpoint, failed and been repaired, it notifies p2 that it is a recovering process. Receiving the recovery message from p2, p1 notifies p2 that it is a live process. Then, p2 and p3 elect a recovery leader. If p2 is the recovery leader, it sends a recovery message to p3 for obtaining p3's incarnation number. Receiving p3's response, it collects determinants for p2 and p3 from only all the live process, in this case, p1. However, it is unable to know the RSN of m_1 and m_2 because p3 has the determinant of m_1 and m_2, but is a recovering process, not a live process.
Thus, p2 may not replay m_3 if it receives m_2 and then m_1 in order and p3, be an orphan process because p3 restarts from the checkpoint after p3 has received m_3 from p2.

3.2 Basic Idea

3.2.1 Handling constraints in mobile computing systems

This section describes how to handle the following constraints in our causal logging protocol.

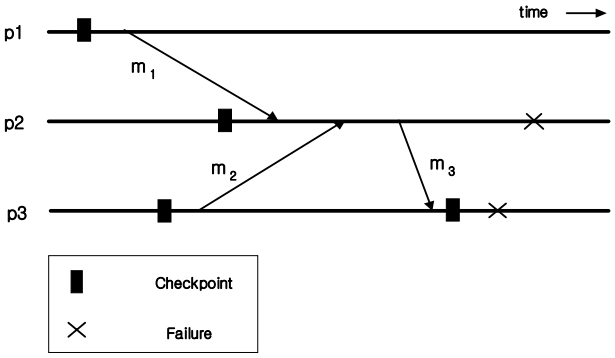

Fig. 2. An execution of three processes with two process failures

• Lack of stable storage: To save its local checkpoint, each MH requires stable storage, However, it is vulnerable to a total catastrophic failure. Thus the MH's disk is not stable. Therefore, each MH uses the stable storage of its local MSS, which the MH currently registers with, for saving its local checkpoint in our protocol.

• Changes in MHs' locations: This constraint complicates the routing of messages. There are many routing protocols for the network layer to handle this constraint [9]. In our protocol, if a MH intends to take a local checkpoint, it sends its local state, the determinants in its volatile memory, its incarnation number and etc. to its local MSS. Incarnation number of each MH is incremented by one whenever the MH fails and starts to recover [18]. When the MSS receives them, it saves them on stable storage and if there was the old checkpoint of the MH, it sends a message for deleting the checkpoint to MSS_{old}, which has saved the checkpoint. The reason is that each MH requires only the latest checkpoint of the MH during recovery in our causal logging protocol and multiple number of checkpoints of the MH need not be saved at MSSs. If a MH enters into a new cell covered by MSS_{new}, the local MSS sends MSS_{new} the identifier of the MSS saving the latest checkpoint of the MH. This approach eliminates the overhead for moving the checkpoint to MSS_{new} whenever the MH enters into MSS_{new}.

• Disconnection: Before disconnecting from its local MSS, MH_p takes a local checkpoint and sends the local MSS its checkpoint and the determinants needed for recoveries of other MHs. Receiving them, the MSS save the checkpoint of the MH_p on stable storage and the determinants in its volatile memory. During the disconnect interval,

application messages destined to the MH_p are buffered at the MSS until the end of the interval. Even if other MHs fail during the disconnect interval, the MSS can provide them the determinants related to them on behalf of the MH_p. When the MH_p reconnects to the local MSS, the MSS sends the buffered messages to the MH_p. Receiving them, the MH_p logs the messages in its volatile memory and delivers them to the application in order. If the MH_p enters into a new cell covered by MSS_{new} during the disconnect interval, MSS_{new} receives the buffered messages from the old MSS and sends them to the MH_p.

3.2.2 Recovery Algorithm

To solve the problems of Elnozahy's recovery algorithm stated in section 3.1, we propose an efficient recovery algorithm as follows: If processes fail, each process requires its last checkpoint and determinants from its local MSS. After having received and restored them, it sends the other processes a recovery message including its incarnation number and state flag respectively for obtaining their incarnation numbers and state flags. The state flag of a recovering process denotes '*R*', and that of a live process denotes '*L*'. If each process receives the recovery message from a recovering process, it sends its state flag and incarnation number to the recovering process. After each recovering process has received the state flags and incarnation numbers of the other processes, it sends the other processes a recovery message with a vector, whose element consists of the state flag and incarnation number of every process, for requiring the determinants related to only itself. Receiving the recovery message from the recovering process, each (live or recovering) process sends the recovering process the determinants related to only it. If any among live processes fails and sends each recovering process a recovery message for obtaining its state flag and incarnation number, the recovering process discards all the determinants collected so far, and restarts collecting the determinants related to only itself from the other processes again. This step ensures that the state of every recovering process is consistent with that of any process even although live processes fail concurrently during recovery. After each recovering process has received the requested determinants from all the other processes, it replays all the received messages in a failure-free execution, completes its recovery, and executes as a live process. Therefore, each recovering process can recover to a consistent state by using its determinants without the stated recovery leader problems and improve asynchrony among recovering processes. Additionally, our algorithm keeps executing live processes in concurrent failures because each live process can immediately decide whether received messages must be discarded or delivered by using its vector including the state flags and incarnation number of every process.

Moreover, our protocol can recover each recovering process to its consistent state if it is integrated with asynchronous checkpointing. To illustrate this fact, consider the example shown in figure 2. In this case, *p2* restores its latest checkpoint and then sends *p1* and *p3* a recovery message for obtaining *p1*'s and *p3*'s incarnation number and state flag in our protocol. After *p3* has received m_3 from *p2*, taken its local checkpoint, failed, been repaired and received the recovery message, it sends *p2* a response including its incarnation number and '*R*'. Receiving the recovery message, *p1* sends *p2* a response including its incarnation number and '*L*'. Then, *p2* receives from *p1* and *p3* their incarnations and flags, and sends *p1* and *p3* a recovery message with *p1*'s and *p3*'s incarnation number and state flag for collecting its determinants respectively. Receiving the message, *p3* can provide *p2* with determinants of m_1 and m_2 including their *RSN*s. Thus, *p2* can replay m_3 and recover to a consistent state after it delivers m_1 and then m_2 to the application by using their *RSN*s.

4. The Protocol for tolerating failures of MSSs

4.1 Basic Idea

To solve the stated problems of the existing replication-based protocols in this paper, we present a receiver-based pessimistic message logging protocol with checkpointing. For example, suppose MH_1 currently obtains a care-of address from MSS_{A1} on network *A* in figure 1. In this case, MH_1 must send a registration request message to MSS_{A1}. Receiving the message like in figure 3, MSS_{A1} saves the recovery information of the message in the stable storage and then, updates the location information of MH_1 using the message. Then, it sends MH_1 a request reply message. This step ensures that one among other MSSs on the network, named MSS_{Ak}, can restore the location information of all the MHs registering with MSS_{A1} by restoring the messages from the stable storage and replaying them even if MSS_{A1} fails. Moreover, each MSS should save the location information of all the MHs registering with it in the stable storage periodically and remove all the logged messages beyond the previous checkpoint from the stable storage. Therefore, this protocol can reduce the failure-free overhead compared with the existing protocols because it needs not forward each registration request message to the other MSSs and wait for each acknowledge message from them. Furthermore, it improves the scalability to a large number of MHs registering with each network compared with the previous protocols because each MSS has only to maintain the location information about the MHs registering with it.

In mobile computing systems, each MSS broadcasts its advertisement message via its wired or wireless network interface every few seconds. Therefore, each MSS can detect if other MSSs fail or not by monitoring their advertisement messages. For example, if MSS_{A1} on network *A* fails in figure 1, live redundant MSSs on the network can detect MSS_{A1}'s failure because they may receive no advertisement message from it. Each MSS has a vector saving a timer of each MSS on the same network. Whenever ADVERTISING_INTERVAL (2~3) seconds has elapsed, it decrements the timer for each MSS by one.

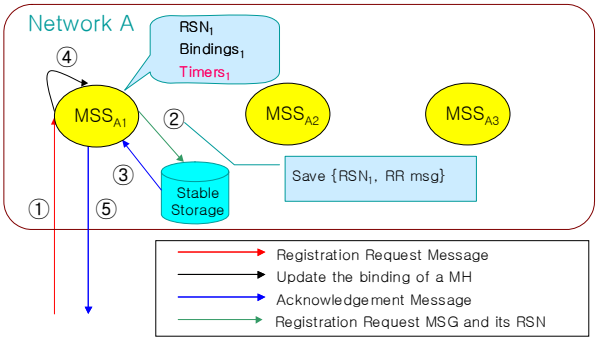

Fig. 3. Message logging procedure

If some MSSs detect MSS_{A1}'s failure, one among them, for example, MSS_{A2} in figure 4 which currently has the minimum number of its managed MHs, takes over MSS_{A1}. This step ensures that the protocol is scalable even during recovery compared with the existing protocols. MSS_{A2} restores the location information about all the MHs registering with MSS_{A1} and obtains the logged messages for MSS_{A1} from the stable storage. Then, it can recover the latest location information about all the MHs, which MSS_{A1} managed before it failed, by replaying the messages in receipt sequence order. After that, MSS_{A2} performs a gratuitous ARP binding MSS_{A1}'s IP address to MSS_{A2}'s hardware address to take over MSS_{A1} [17]. Therefore, the protocol provides the MHs with the transparency of their MSSs' failures and replacement.

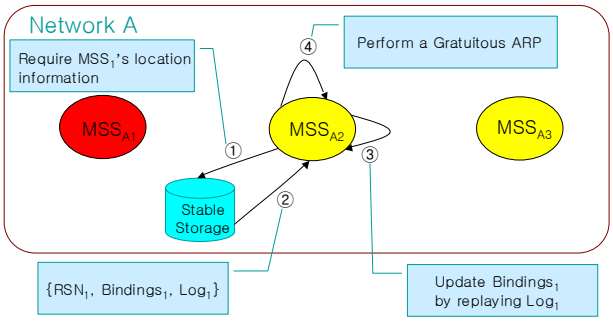

Fig. 4. Takeover procedure

If a failed MSS MSS_{A1} is repaired like in figure 5, it should monitor any advertisement message, including its IP address, for a few seconds and perform a gratuitous ARP mapping its IP address to its hardware address. If no other MSS took over it, it should restore the location information about all the MHs registering with it and obtain the logged messages for it from the stable storage. Then, it can recover the latest location information about the MHs before it failed by replaying the messages in receipt sequence order. If it receives an advertisement message including its IP address from a live MSS MSS_{A2} in figure 5, it should require from MSS_{A2} the location information of all the MHs registering with it. If MSS_{A2} fails during its recovery, the recovering MSS MSS_{A1} can recover the latest location information about all the MHs registering with it by using its checkpoint and logged messages on the stable storage.

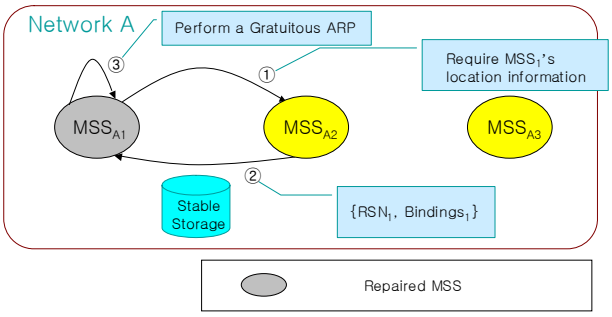

Fig. 5. Recovery procedure

5. Discussion

The asynchronous checkpointing algorithm proposed by Achrarya et al. [1] requires each process to take a checkpoint whenever a message transmission precedes a message reception. This approach might force each process to take as many checkpoints as the number of messages if the message reception and transmission are interleaved. It may results in high failure-free overhead. Although some checkpointing algorithms [14, 19, 21, 22] reduce the number of checkpoints to be saved on stable storage, to ensure correctness, each process still needs to take forced checkpoints when required and keep much more checkpoints in them than in our checkpointing algorithm. Our causal logging protocol never requires each process to take the forced checkpoints. The adaptive checkpointing protocol proposed by Neves and Fuchs [15] uses time to indirectly coordinate the creation of recoverable consistent checkpoints. It requires that checkpoints be sent back only to Home Agents. However, our protocol has not the limitation because each process

takes its local checkpoint on the stable storage of its current local MSS in the protocol. Yao et al. [20] propose a receiver-based pessimistic message logging protocol for tolerating failures of MHs. It allows failed processes to recover fast and asynchronously. However, it has the common disadvantages of pessimistic logging protocols. Thus, it may result in high failure-free overhead. Although our protocol may recover failed processes slower than that of Yao et al., it results in lower failure-free overhead than that of Yao et al. Next, we will compare our protocol with other causal logging protocols. FBL protocols [2, 3] block the execution of live processes during recovey because they should prevent the live processes from becoming orphan processes. Manetho protocol [6] can recover concurrently failed processes but it requires live processes to refrain from accepting certain application messages during recovery and some synchronous logging to stable storage while recovery is ongoing. Elnozahy's recovery algorithm [4] can recover concurrently failed processes and it continues executing live processes during recovery. However, it requires a central recovery leader, which may be a performance bottleneck. Moreover, it results in nontrivial election overhead and if the leader fails continuously before it completes its recovery procedure, the other recovering processes should continue being blocked. Additionally, if it were integrated with asynchronous checkpointing, it could results in inconsistency problems in case of concurrent failures like in figure 2. Our protocol solves Elnozahy's problems and improves asynchrony among recovering processes because each recovering process is responsible for only its recovery.

In next part, we intend to compare our receiver-based pessimistic logging protocol (denoted by ORPP) with the protocol presented in [8] (denoted by FTMIPS) briefly. Generally, performance indices used for evaluation of scalability of FTMIPS and ORPP are the number of MHs whose location information is managed by each MSS on a network (denoted by *NOMHMSS*) and the latency time for its processing a registration request message from each MH (denoted by *LTRR*). Table 1 shows the parameters, which *NOMHMSS* depends on, and their meanings. For simplicity, we suppose that the same number of MHs registers with each MSS on a network. It means that if *NOMH* is n and *NOMSS* is m, the number of MHs registering with each MSS is (n / m).

Table 1. Parameters and their meanings

NOMH	Number of MHs registering with a network
NOMSS	Number of redundant MSSs on a network

First, we evaluate scalability provided by ORPP and FTMIPS with respect to *NOMHMSS* during failure-free operation. If *NOMHMSS* of the two protocols are calculated using the parameters respectively, that of ORPP is (*NOMH* / *NOMSS*) whereas that of FTMIPS is *NOMH* during failure-free operation. The reason for these results is that each MSS on a network must maintain the location information about the MHs registering with all the redundant MSSs on the same network in FTMIPS whereas each MSS has only to maintain the MHs registering with it in ORPP. Therefore, we can see that ORPP improves scalability to a large number of MHs managed by each MSS compared with FTMIPS during failure-free operation. Next, we evaluate scalability provided by ORPP and FTMIPS with respect to *LTRR* during failure-free operation. If each MSS receives a registration request message from a mobile host in FTMIPS, it should process the message and forward the message to its peers and wait for receiving all the acknowledgement messages from them. Thus, the total number of messages generated per registration request message in FTMIPS is ($2 \times$ *NOMSS*) and the number of messages on the critical path is (*NOMSS* + 1). However, in ORPP, it should process the message and send a stable storage server a message for saving the recovery information of the message to stable storage and wait for receiving an acknowledgement message from it. Thus, the total number of messages generated per registration request message in ORPP is 2 and the number of messages on the critical path is 2.

Next, we evaluate the overhead of the two protocols for taking over or recovering failed MSSs. In FTMIPS, a live MSS can recover failed MSSs fast because it always maintains the location information about all the MHs registering with not only itself but also its peers. The takeover time of ORPP may be longer than that of FTMIPS because each live MSS has only to maintain the location information about the MHs registering with it and should recover the location information of failed MSSs from the stable storage.

If there are live MSSs on a network, the recovery time of failed and repaired MSSs are the same in the two protocols because they can recover their location information from the live MSSs in both. However, if not so, each failed MSS can recover its location information from the stable storage in ORPP whereas it can not recover anywhere in FTMIPS.

6. Conclusion

In this paper, we first identified the limitations raised when traditional causal message logging protocols are applied to mobile computing systems and their problems such as blocking the execution of live processes during recovery, requiring a central recovery leader among recovering processes and making inconsistency if

Elnozahy's protocol were integrated with asynchronous checkpointing. Then, we presented a causal message logging protocol for tolerating failures of MHs. It can efficiently handle all the stated constraints in mobile computing systems and allow processes to make consistency with each other even in concurrent failures when it is integrated with asynchronous checkpointing. Moreover, it enables live processes to execute their computation even in concurrent failures and improves asynchrony among recovering processes because each recovering process is responsible for only its recovery using its vector consisting of incarnation number and state flag of every process.

Second, the existing replication-based protocols for tolerating failures of MSSs result in high synchronization overhead among replicas and are not scalable. Therefore, we presented a receiver-based pessimistic message logging protocol for solving the problems of the existing protocols. We showed that it reduces the failure-free overhead and improves scalability to a large number of MHs registering with each network compared with the existing protocols in section 5. Additionally, we can see that even if all the MSSs on a network fail, they can recover all the related information of the MHs registering with the network after they have been repaired and executed our protocol.

References

[1] A. Acharya and B. R. Badrinath. "Checkpointing Distributed Applications on Mobile Computers," *Proc. the 3rd Int'l Conf. On Parallel and Distributed Information Systems*, Sep. 1994.

[2] L. Alvisi, B. Hoppe, and K. Marzullo. "Nonblocking and Orphan-Free Message Logging Protocols," *Proc. the 23rd Fault-Tolerant Computing Symposium*, pp. 145-154, June 1993.

[3] L. Alvisi and K. Marzullo. "Message Logging: Pessimistic, Optimistic, Causal and Optimal," *IEEE Transactions on Software Engineering*, 24(2): pp. 149-159, Feb. 1998.

[4] E. N. Elnozahy. "On the relevance of Communication Costs of Rollback Recovery Protocols," *Proc. the 15th ACM Symposium on Principles of Distributed Computing*, pp. 74-79, 1995.

[5] E. N. Elnozahy, L. Alvisi, Y. M. Wang and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, 34(3), pp. 375-408, 2002.

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

[7] O. P. Damani and V. K. Garg. "How to Recover Efficiently and Asynchronously when Optimism Fails," *Proc. the 16th International Conference on Distributed Computing Systems*, pp. 108-115, 1996.

[8] R. Ghosh and G. Varghese. Fault-Tolerant Mobile IP. Technical Report WUCS-98-11, Washington University, April 1998.

[9] J. Ioannidis, D. Duchamp, and G.Q. Maguire. "Ip-based Protocols for Mobile Internetworking," *Proc. Of ACM SIGCOMM Symp. on Communication, Architectures and Protocols*, pp. 235-245, Sep. 1991.

[10] D. B. Johnson and W. Zwaenpoel. "Sender-Based Message Logging," *In Digest of Papers: 17 Annual IEEE International Symposium on Fault-Tolerant Computing*, pp. 14-19, June 1987.

[11] D. B. Johnson and W. Zwaenepoel. "Recovery in distributed systems using optimistic message logging and checkpointing," *Proc. the 7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 171-181, August 1988.

[12] P. Krishna, N. H. Vaidya, and D. K. Pradhan. "Recovery in Distributed Mobile Environments," *IEEE Workshop on Advances in Parallel and Distributed System*, Oct. 1993.

[13] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7), pp. 558-565, 1978.

[14] D. Manivannan and Mukesh Singhal. "A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing," *Proc. the 16th International Conference on Distributed Computing Systems*, pp. 100-107, 1996.

[15] N. Neves and W. K. Fuchs. "Adaptive Recovery for Mobile Environments," *Communications of the ACM*, 40(1): pp. 68-74, Jan. 1997

[16] C. Perkins. IP Mobility Support. RFC 2002, October 1996.

[17] D. C. Plummer. An Ethernet Address Resolution Protocol - or – Converting Network Protocol Address to 48 bit Ethernet Address for Transmission

on Ethernet Hardware. STD 37, RFC 826, November 1982.

[18] R. B. Strom and S. Yemeni. “Optimistic recovery in distributed systems,” *ACM Transactions on Computer Systems*, 3(3): pp. 204-226, Apr. 1985.

[19] Y. Wang and W. K. Fuchs. “Lazy Checkpoint Coordination for Bounding Rollback Propagation,” *Proc. the 12th Symposium on Reliable Distributed Systems*, pp. 78-85, 1993

[20] B. Yao, K. –F. Ssu and W. K. Fuchs. “Message Logging in Mobile Computing,” *Proc. 29th Annual IEEE International Symposium on Fault-Tolerant Computing*, pp. 14-19, Oct. 1999.

[21] G. Cao, M. Singhal, “Checkpointing with mutable checkpoints”, *Theoretical Computer Science*, Vol. 290, pp. 1127-1148, 2003

[22] M. Chaoguang, W. Dongsheng and Z. Yunlong, “An Efficient Computing-Checkpoint Based Coordinated Checkpoint Algorithm”, *Lecture Note In Computer Science*, Vol. 4096, pp. 99-109, 2006.

JinHo Ahn received his B.S., M.S. and Ph.D. degrees in Computer Science and Engineering from Korea University, Korea, in 1997, 1999 and 2003, respectively. He has been an Assistant Professor in department of Computer Science, Kyonggi University. His research interests include distributed computing, fault-tolerance, mobile computing systems, mobile agent systems and sensor networks.