Performance of K-Fault Tolerant Checkpointing for Mobile Agents

Taesoon Park

***Abstract*—This paper studies the performance of two fault tolerant schemes for mobile agents. The behavior and the performance of the k-fault tolerant checkpointing scheme are compared with those of the replication schemes. To study the performance of two schemes, an experimental system has been built on the Aglets mobile agent system. Also a simulation system has been built to further study the performance of two schemes under various failure environments. By comparing two schemes, the system environments where the k-fault tolerant checkpointing scheme works well and the system parameters which mainly affect the performance are investigated.**

***Index Terms*—fault tolerance, mobile agent system, failure-recovery, k-fault tolerant checkpointing, agent replication .**

I. Introduction

A mobile agent is a software program which migrates from a site to another site to perform a task given by a user [1]. The execution of a mobile agent is autonomous and asynchronous. Once an agent is launched by a user, it decides its migration and execution by itself; and no more interaction with the user is required. For autonomous and asynchronous execution, the agent carries program codes, the data required for the computation and the data collected from visiting sites. Also, it carries intermediate execution states for the continuous execution throughout the sites.

Because of these characteristics, a mobile agent is known to be suitable for the applications which should run in the environment with low network resources. For example, the system with the low network bandwidth or the system which requires frequent disconnection from the network is considered. The mobile agent has drawn an attention as a new distributed computing paradigm and many mobile agent systems have been developed to support execution and migration of the agents.

For the mobile agent system to support the agents in various application areas, the issue on the reliable agent execution should be considered in depth. While executing on a site or migrating between the sites, an agent may get lost due to a system failure. Also, while the failure and recovery of the related sites are progressing, the execution of an agent may get duplicated unless the execution of an agent is carefully coordinated between the sites. Reliable execution of a mobile agent is to guarantee the exactly-once execution of an agent even in the case of a system failure.

As the importance of reliable agent execution is emphasized, many fault tolerance schemes have been proposed for mobile agent systems, which are categorized into the replication schemes [2],[3],[4] and the checkpointing schemes [5],[6],[7]. For each stage of execution, an agent in the replication scheme is replicated and sent to a set of sites so that a replicated agent which survives the failure can take care of the recovery action in case a failure of some sites. Also, to eliminate the duplicate execution, execution of the replicated agents is processed with a distributed transaction [8], the leader election procedure [2] or the agreement procedure [3],[4] at the end of each execution stage.

However, the replication scheme may not work efficiently for the environment where the agent size is large or the network cost is high, since the cost of agent replication is not negligible. Hence, in the checkpointing scheme, an agent is sent to a single site. Instead, the state of the agent is checkpointed into a stable storage on the arrival of each site so that the agent can be re-executed from the checkpointed state in case of a system failure. Or, as in the k-fault-tolerant checkpointing scheme [6], an agent takes a checkpoint at the end of each stage and it does the recovery action using these checkpoints taken on the sites previously visited by an agent. To detect the failure of the agent and manage the checkpoint, an observer agent is created on each stage. In case of a site failure, the agent can recover from one of the previous checkpoints so that the system can achieve a higher degree of reliability.

Also, some fault tolerant schemes for the cooperative agents have been proposed in [9], [10]. In this case, the failure of an agent may affect the computation of other cooperating agents and hence, the computational dependency among the agents should be traced and the related agents should take the recovery action, together. In any case, the underlying recovery action follows the replication or the checkpointing scheme.

In this paper, we study the behavior of the replication and the checkpointintg schemes in more detail and compare the performance of the k-fault tolerant checkpointing scheme and the replication schemes in various system environments. To study the performance of two schemes, an experimental system has been built on the Aglets mobile agent system [11]. Also a simulation system has been built to further study the performance of k-fault tolerant checkpointing scheme under

Manuscript received December 10, 2012. Taesoon Park is with the Department of Computer Engineering, Sejong University, Seoul, 143-747, KOREA (phone: +822-3408-3240; fax: +822-3408-3321; e-mail: tspark@ sejong.ac.kr).

various failure environments. It is known that the checkpointing scheme requires the low overhead compared to the replication scheme during the normal operation. However, considering the concurrent failure environments, the computation loss under the checkpointing scheme can be slightly larger than that of replication schemes. By comparing the performance of two schemes, the system environments where the k-fault tolerant checkpointing works well and the system parameters which mainly affect the performance of the k-fault tolerant checkpointing are investigated.

The rest of this paper is organized as follows: Section 2 presents the mobile agent system model. In Section 3, the k-fault-tolerant checkpointing scheme and the replication scheme are described. The performance of the k-fault tolerant checkpointing scheme is compared with the performance of the replication schemes in Section 4. Section 5 concludes the paper.

II. Mobile Agent System Model

A mobile agent is a set of program codes, which is executed in a system site and moves from a system site to another site. A mobile agent system consists of a number of system sites connected by a communication network, and each site provides one or more *places* to support execution and migration of mobile agents. A place is responsible for preparing migration of an agent, migrating the agent, registering a new agent, and providing the access point to the local resources. In other words, an agent executes its task on the place and migrates between the places.

A mobile agent migrates from a place to another place on the same site or on the different system sites. While residing in a place, the agent performs a given task by accessing the local resources of the site. Also, the agent may communicate with another agent in the same site or in some remote site. The execution of an agent in a place is called a *stage*, and the lifecycle of an agent consists of a series of stages and the migration between the stages. We denote a mobile agent with an identifier i as MA_i and the a-th stage of MA_i as $SG_{i,a}$. The stage $SG_{i,a}$ means the execution of a task after the a-th place migration and we also denote the agent in $SG_{i,a}$ as $MA_{i,a}$.

Failures considered for an agent are the *agent failure*, the *place failure* and the *system failure*. For all of these failure types, the *fail-stop* failure model [12] is assumed. According to this model, a failed component immediately stops its execution and does not perform any malicious actions to the other components. The failures are also assumed to be transient and independent. That is, once the failed component recovers from a failure, the same type of failure does not likely occur, again; and the failure of one component is not related to the failure of another component.

Once an agent stops the execution due to any of the above-mentioned failures, it has to be restarted since the agent loss in the fault-tolerant system should not be allowed even in case of a failure. Agent and place failures are usually caused by incorrect agent states, place states or system states. In case of a system failure, the agent execution state saved in the volatile memory may get lost. Hence, for any failure type, an agent has to restart the execution from an earlier state. To prevent the restart from the initial state, the checkpointing or the replication scheme is used.

III. Fault Tolerant Schemes for Mobile Agents

In this section, two fault tolerant schemes for reliable mobile agent systems are described and compared. One is the k-fault tolerant checkpointing and the other is the replication scheme.

A. Checkpointing

Checkpointing is an operation to save the intermediate state of a process into a stable storage so that the process can recover from the saved state after a system failure occurs. For the reliable mobile agent system, the checkpointing scheme can also be used. One way to implement the checkpointing is that an agent takes a checkpoint of its current state on the arrival of each new place. An advantage of this approach is that the recovery of an agent can be localized. In other words, the failure of a system site does not force the recovery of other sites. However, in this approach, an agent can recover only after the system site recovers from the failure, in case of a system failure. Therefore, an agent may get blocked for a long time until the system site recovers from a failure. In the worse case, the agent may get lost if the system site cannot be recovered.

Another way is to take a checkpoint of an agent at the end of each stage. In this case, the checkpoint is used when the next execution site fails. A mobile agent moves from a site to another site to complete the assigned task. When the current visiting site of the agent fails, the agent is blocked or lost in the worse case. However, most of the agent owners may want their agents to skip the problem site and continue the remaining trips. Or, the owners may want the agent to come back to the initial site if it cannot successfully complete the assigned task. In either case, the failure of a certain site needs to be detected and the agent should be able to reroute their remaining trip. For this purpose, the checkpoint is taken at the end of each stage, so that the agent can be recovered and re-routed from there when the failure of the next execution site is detected.

B. K-Fault Tolerant Checkpointing

When the checkpoint is taken at the end of each stage, an observer agent is required to detect the failure of the next execution site and recover the agent from the saved checkpoint. In most of the mobile agent systems, an agent is first replicated and the replicated agent is transferred to the next execution site using the *Remote Method Invocation*, for the agent migration. When the migration is successful, the agent remained in the previous site is eliminated. However, in the k-fault tolerant checkpointing scheme, the agent remained in the previous site is used as an observer agent instead of being eliminated. The observer manages the checkpoint of the agent and detects any possible failure of the next execution site. Figure 1 shows an example of agent migration and checkpointing. As shown in the figure, before the mobile agent, $MA_{i,a}$, migrates for the next

stage, $SG_{i,a}$, it first takes a checkpoint and is then replicated to create an observer agent, $OA_{i,a}$.

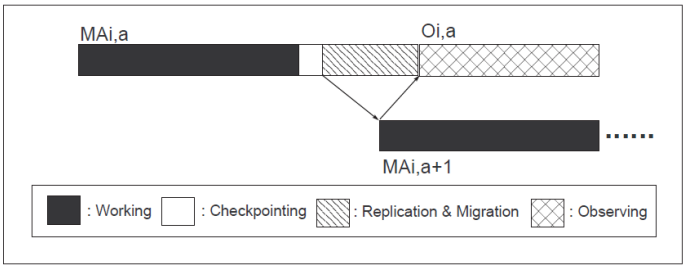

Fig. 1 Checkpointing and Observer Creation

An observer agent may recover the agent from the checkpointed state when it detects a failure of the current execution site. However, if there are concurrent failures on both sites, both of the agent and the observer agent may be lost. Therefore, to tolerate concurrent failures of up to *k* sites, *2k+1* agents are required. Among these, one is the primary agent which is responsible for the initial execution of the task on each stage and *k* observer agents are required to manage *k* checkpoints and recover the agent even after *k* concurrent failures. Also, *k* more observer agents are used to reach the consensus, in case that more than one observer agent try the recovery action caused by false detection of the failure of other sites. Hence, in the k-fault tolerant checkpointing scheme, *2k* observer agents are created to take care of the recovery action along the agent's traveling path.

To agree on the completion of a stage $SG_{i,a}$ by $MA_{i,a}$ and eliminate any duplicate execution by other observers, the primary agent and the observers perform the consensus procedure at the end of each stage. If any observer falsely detects the failure of the primary agent, it is possible that there can be duplicate execution by the primary agent and the observer agent for the same task. To guarantee no duplicate execution, $MA_{i,a}$ sends out the $M\_exec\_end_{i,a}$ messages to the observers before the migration, and on the receipt of the $M\_exec\_end_{i,a}$ message, an observer agent sends back the acknowledgment message, $M\_ack_{i,a}$ to the primary agent. When the primary agent, $MA_{i,a}$, receives the $M\_ack_{i,a}$ messages from the majority of the observers, it confirms that there is no duplicate execution for the stage, $SG_{i,a}$ and sends back the $M\_confirm_{i,a}$ messages to the observers. $MA_{i,a}$, then migrates for the next stage.

An observer agent can eliminate the checkpoint after it completes the *k+1* consensus and it can be terminated after it participates in the *2k+1* consensus.

C. Failure-Recovery

To detect the failure of the primary agent, the time-out mechanism is used. An observer agent sets a timer with a certain time-out value for each stage, $SG_{i,a}$. When the observer receives the $M\_exec\_end_{i,a}$ message before the timer expires, it resets the timer for the next stage, knowing that there is no failure during the stage, $SG_{i,a}$. However, if the timer expires without receiving the $M\_exec\_end_{i,a}$ message, the observer suspects the failure of the primary agent, $MA_{i,a}$ and begins the consensus procedure. For the consensus, any observer which first obtains the votes from the majority of the other observers can be elected to take care of the recovery action, as proposed in [2]. Or, as proposed in [3], the observer with the highest priority can take care of the recovery. When the primary fails, the computation performed by the agent should be lost and the recovery from the latest checkpoint can minimize the computation loss. Hence, it is natural to follow the consensus approach described in [3].

Figure 2 briefly describes the consensus process proposed in [3]. As shown in the figure, when an observer cannot receive the $M\_exec\_end_{i,a}$ message, it first sends the *Nack* message to the primary agent. If the observer is the one with the highest priority, it waits for the *estimate* message from other observers. Otherwise, if the observer is not the one with the highest priority, it sends the *estimate* message to the observer with the highest priority. The *Nack* messages are to inform the primary agent of the failure detection. In case that the primary does not fail and it just executes slowly, it can send the *alive* message to the observers. However, if it cannot send the *alive* message before it receives the *Nack* message from the majority of the observers, it is assumed to fail.

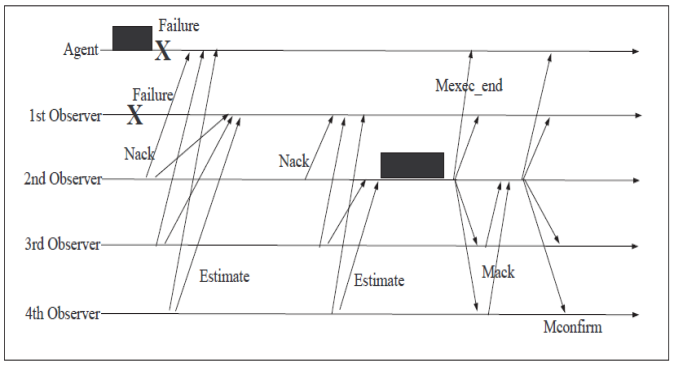

Fig. 2 Consensus Procedure

The *estimate* messages are to inform the observer agent with the highest priority of the *permission* of the recovery action. Therefore, the observer agent can recover the primary agent from the checkpointed state, when it receives the *estimate* messages from the majority of the observers. Note that the *Nack* message and the *estimate* message have the effect only when they are confirmed by the majority observers. Using the majority consensus, any false detection of a failure can be ignored. Once an observer sends out an *estimate* message, it resets the time-out timer for the failure detection of the new primary. When the time-out timer expires, the observers perform the same process. Figure 2 shows the case that there are concurrent failures of both of the primary agent and the observer agent with the highest primary. After two rounds of the consensus process, the observer agent with the second priority takes over the recovery action.

D. Replication

Fault tolerant schemes supported by agent replication can be implemented in various ways as proposed in [4]. In the

replication scheme, a primary agent makes *2k* replicas for each stage of the task and migrates the replicas to the *2k* different sites. As for the k-fault-tolerant checkpointing scheme, *k* replicas are made to detect the failure of the primary agent and perform the alternative execution for the current stage. The rest of the replicas are made to participate in the consensus at the end of each stage. Therefore, the primary agent with *2k* replicas can tolerate up to *k* concurrent failures.

In the synchronous replication scheme, migration of the replicas is processed in two phases. In the first phase, the replicas for the next stage are made, the execution site for each replica is determined and then the replicas are sent by the primary agent. For each replica migration, the *ACK* message should be replied back. In the second phase, the primary agent migrates itself for the next stage after receiving the *ACK* messages for all of the replicas. Since in the synchronous replication scheme, the primary agent takes care of the replication and migration, the time taken for the replication should be long, especially when the agent size is large and the number of replicas is high.

Compared to the synchronous replication scheme, the task execution of a stage by the primary agent and the replica migration of the same stage by the previous primary agent are overlapped in the asynchronous replication scheme. The first replica migrated by the previous primary agent becomes the primary for the new stage and begins the execution without waiting for the complete migration of other replicas. While the new primary executes the task of the new stage, the previous primary keep replicating and migrating the rest of the replicas. As in the synchronous scheme, the previous primary is terminated when it receives the *ACK* messages from all of the migrated replicas.

In this scheme, if the working time of the primary agent for the new stage is longer than the migration time of all the replicas, the agent replication time can completely be masked. However, if the replica migration time is longer, the primary agent of the new stage has to wait for the consensus process. As a result, the consensus time should be longer when the working time is relatively short or the replica migration time is long. One way to reduce the replica migration time is the use of consensus agents. As mentioned before, among the *2k* replicas, only *k* replicas are used for the failure detection and the alternative execution. The rest *k* replicas are only to participate in the consensus. Therefore, instead of using *2k* replicas, *k* consensus agents can be used with the *k* replicas. The consensus agent contains only the simple codes to perform the consensus and hence the migration time of the consensus agent can be much less.

IV. Performance of K-Fault Tolerant Checkpointing

A. Performance of Failure-free Operation

To measure the cost of fault tolerant mobile agents during the failure-free execution, k-fault-tolerant checkpointing and two replication schemes have been implemented on top of the Aglet system. The Aglet is a Java-based mobile agent system and for our experiments, *Aglet DSK 1.1b2* was used. For an agent to access the appropriate fault-tolerance methods, the *Replication class* was created under the *AgletClass*. Two replication classes and one checkpointing class were created under the Replication class. Also, to observe the influence of the agent size, we implemented a *Sizer class*, which iteratively inserts garbage values in the Vector class so that the size of the serialized agent object can be controlled.

A cluster of forty Pentium IV 1 GHz PCs connected through a 100 Mbps Ethernet was used for the experiments. Each machine supported an *AgletContext* and an agent traversed the sites in a predetermined order. For the execution sites of the agent and the replicas not to be overlapped, we used logical grouping of PCs and for the replication schemes, the sites to execute the replicas were randomly selected from different groups. A task assigned to an agent consists of eight stages and at each stage, the agent sleeps for *W* seconds instead of performing any action. For the stable performance, twenty runs of the agent task were measured and then the measured values were averaged out.

Figure 3 shows the performance of three fault-tolerant schemes when the sizes of an agent are 100 KBytes and 1000 KBytes. For this result, 1000 ms. working time is used for one stage and one primary agent and four observers (or replicas) are used for each stage. As it is expected, k-fault tolerant checkpointing (denoted by KCP) requires the least overhead. The synchronous replication scheme (denoted by S) spends most of the execution time for the replication. The asynchronous replication scheme (denoted by A) spends most of the execution time for the consensus, since the consensus can be performed concurrently with the replication.

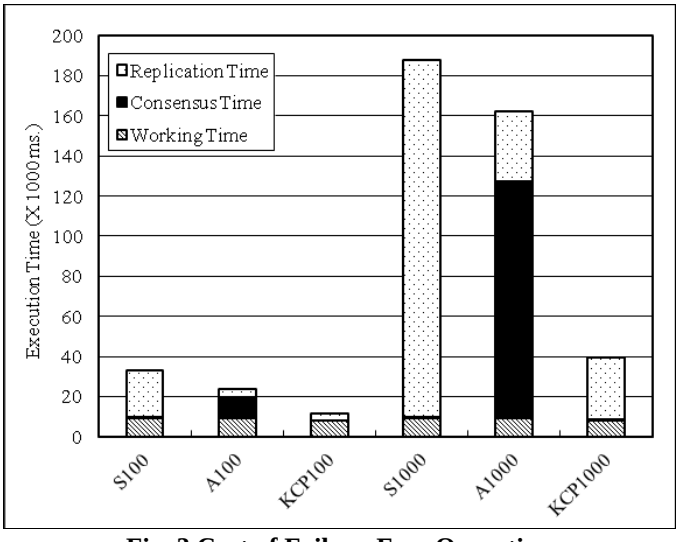

Fig. 3 Cost of Failure-Free Operation

As the agent size becomes 10 times larger, the execution times of the scheme S and the scheme A become 5.65 times and 6.62 times longer, respectively. The performance of the scheme A becomes worse as the agent size increases. The scheme A achieves 32.9% reduction of the execution time compared to the scheme S when the agent size is 100 KBytes. However, it shows 21.5% reduction when the agent size is 1000 KBytes. The main

reason of this performance is that the migration time of replicas cannot fully be masked by the execution time of the primary when the agent size is large.

Compared to the replication schemes, scheme KCP shows the least cost increases when the agent size becomes 10 times larger and also shows a very stable performance. When the agent size is 100 KBytes, the scheme KCP scheme achieves 69% reduction of the execution time compared to the scheme S while it achieves 79% reduction when the agent size is 1000 KBytes. Therefore, we can conclude that the performance gain of the scheme KCP is higher as the agent size increases, unlike the other replication schemes.

B. Performance of Failure-recovery Operation

To further evaluate the cost of k-fault-tolerant checkpointing scheme in the failure environment, we also have built a simulation system. The simulation system exactly follows our experimental system regarding the behavior of the agent under the k-fault tolerant checkpointing and the replication schemes. The system parameters used for the simulation are as follows:

The working time of an agent for one stage is 1000 ms.; the checkpointing time of an agent is 10 ms.; the replication time of an agent is also 10 ms.; and the transfer delay of the consensus-related messages is 5 ms.. These time values were obtained from our experimental system when the size of the agent is 100 KBytes. For the simulation, these values were used as mean values and it was assumed that each of the time intervals follows an exponential distribution. The behavior of the agent has been simulated with various values of the failure interval and the time-out interval, each of which is also assumed to follow an exponential distribution. To obtain the stable performance results, the overhead of the agent migrating 1000 sites has been measured.

Figure 4.(a) shows the execution time of the agent employing k-fault-tolerant checkpointing, as the failure rate varies from 0.000001 to 0.0002. For each failure rate, three time-out values, 10000 ms., 5000ms., and 2000ms., were used. The label, *T=X*, in the figure denotes that the time-out value is *X* ms.. As it is expected, the execution time increases as the failure rate becomes higher, since the higher failure rate may cause more concurrent failures. In that case, an observer with the lower priority may take care of the recovery action and hence the larger amount of tasks has to be redone. The execution time is more sharply increased when the time-out value is large, since the longer time-out period may cause more delay for failure detection.

Figure 4.(b) compares the failure detection time and the computation loss caused by the failures, when the failure rate is 0.0001. The computation loss is the amount of the task which has to be re-done due to the failure and therefore it is not affected by the time-out value as shown in the figure. However, as it is expected, the time-out value heavily affects the failure detection time. When the time-out value is large, detection of the primary's failure takes a longer time and the overall execution time of the agent becomes longer. As it is shown in Figure 4.(a) and (b), using the short time-out value makes it possible to detect any failure faster.

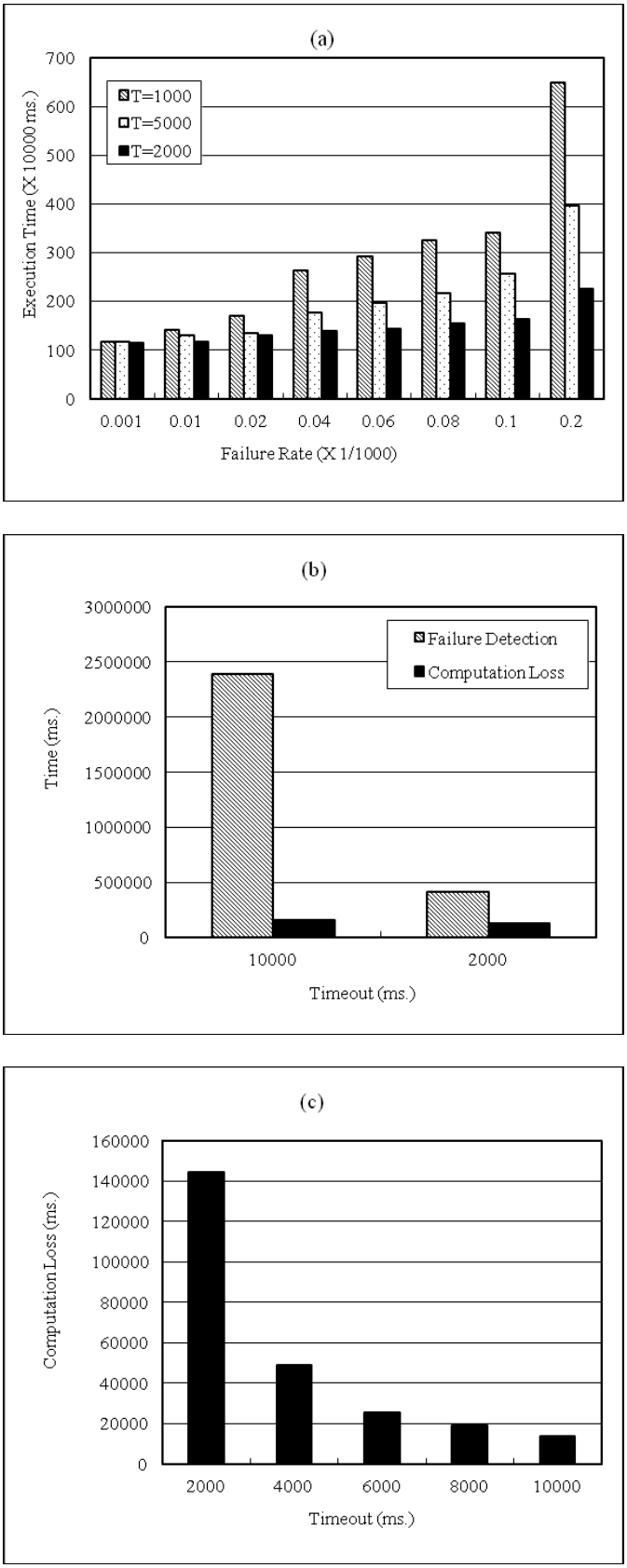

Fig. 4 Cost of Failure-Recovery Operation

However, the short time-out value causes another problem,

which is the false failure detection. If the time-out value is too short, there is a high probability that an observer may falsely detect the failure of the primary agent. This case happens when the primary agent receives *estimate* messages from the majority of the observer agents before it sends back the *Nack* message. Then, the observer may take over the task of a stage even though the primary is still alive but it is slowly processing. Figure 4.(c) shows the computation loss due to the false failure detection when there is actually no failure in the system.

The computation loss due to the short timeout value is the common problem for all the fault tolerant schemes. However, it can be a more serious problem for the k-fault tolerant checkpointing scheme compared to the replication schemes. Figure 5 shows the computation loss under various failure rates when a very short timeout value is used. For this performance results, 2000 ms. is used for the mean timeout value, which is relatively short compared to the working time of 1000 ms.

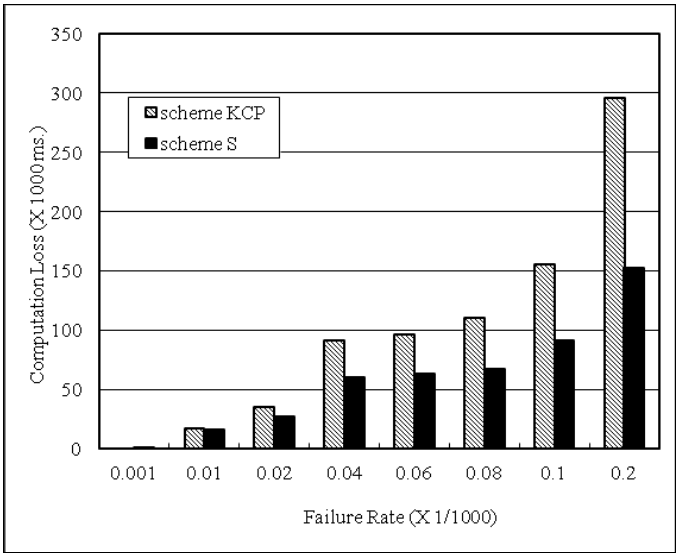

Fig. 5 Cost of Concurrent Failures

Figure 5 compares the computation loss of the k-fault tolerant checkpointing scheme and the synchronous replication scheme. When an agent fails, it logically recovers from the beginning of the current stage in the replication scheme, even if the recomputation should be performed on another site. Hence, a short timeout value may cause the iterative re-execution of the current stage. However, in the k-fault tolerant checkpointing scheme, an agent may have to restart from an earlier checkpoint if there are concurrent failures. When the failure rate is high, there can be the high possibility of concurrent failures of the site performing the current stage and the earlier site keeping the most recent checkpoint. Or, since the timeout value is very short, the observers may falsely detect concurrent failures of the sites.

In both cases, the computation loss of the k-fault tolerant checkpointing scheme becomes larger as shown in the figure, and the performance degradation may be a serious problem when the failure rate is high. Besides the computation loss, the failure-related performance of the k-fault tolerant checkpointing and the replication scheme are very similar.

V. CONCLUSIONS

In this paper, the performance of k-fault tolerant checkpointing scheme has been compared with the performance of the replication schemes. A checkpointing scheme requires the low overhead compared to the replication scheme during the normal operation. Especially when the agent size is large and the replication time takes longer, the k-fault tolerant checkpointing scheme shows the superior performance. However, when the concurrent failures are considered, the computation loss under the checkpointing scheme can be larger than that of replication scheme. Especially when the timeout value is relatively short compared to the working time, performance degradation of the k-fault tolerant checkpointing scheme cannot be negligible. Therefore, it is very important to select the appropriate timeout value when the failure rate is high.

REFERENCES

[1] J. Baumann, F. Hohl, K. Rothermel and M. Strasser, "Mole - Concepts of a Mobile Agent System," World Wide Web Journal, Vol. 1, No. 3, pp. 12-137, 1998

[2] M. Strasser and K. Rothermel, "Reliability Concepts for Mobile Agents," International Journal of Cooperative Information Systems, Vol. 7, No. 4, pp. 355-382, 1998.

[3] S. Pleisch and A. Schiper, "FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach," Proc. of the Int'l Conf. on Dependable Systems and Networks, pp. 215-224, 2001.

[4] T. Park, I. Byun, H. Kim and H.Y. Yeom, "The Performance of Checkpointing and Replication Schemes for Fault Tolerant Mobile Agent Systems," Proc. of the 21st Symp. on Reliable Distributed Systems, pp. 256--261, 2002.

[5] M. Strasser and K. Rothermel, "System Mechanism for Partial Rollback of Mobile Agent Execution," Proc. of the 20th Int'l Conf. on Distributed Computing Systems, pp. 20-28, 2000.

[6] T. Park and J. Youn, "The K-Fault-Tolerant Checkpointing Scheme for the Reliable Mobile Agent System, Proc. of the 5th Int'l Conf. on Parallel and Distributed Computing: Applications and Technologies, pp. 577-581, 2004.

[7] J. Yang, J. Cao and W. Wu, "CIC: An Integrated Approach to Checkpointing in Mobile Agent Systems," Proc. of the 2nd Int'l Conf. Semantics, Knowledge and Grid, pp. 1-4, 2006

[8] H. Vogler, T. Kunkelmann and M.L. Moschgath, "An Approach for Mobile Agent Security and Fault Tolerance using Distributed Transaction," Proc. of the Int'l Conf. on Parallel and Distributed Systems, pp. 268-274, 1997.

[9] M.A.J. Jamali and H.E. Shabestar, "A New Approach for a Fault Tolerant Mobile Agent System," in Proc. of the 12th ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, networking and Parallel/Distributed Computing," pp. 133-138, 2011

[10] R. Singh and M. Dave, "Antecedence Graph Approach to Checkpointing for Fault Tolerance in Mobile Agent Systems," IEEE Transactions on Computers, Vol. 62, No.2, pp. 247-258, 2013

[11] G. Karjoth, D.B. Lange and M. Oshima, "A Security Model for Aglets," IEEE Internet Computing, Vol. 1, No. 4, pp. 68-77, 1997.

[12] R.D. Schlichting and F.B. Schneider, "Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems," ACM Transactions on Computer Systems, Vol. 1, No. 3, pp. 222--238, 1983.