Lazy Agent Replication and Asynchronous Consensus for the Fault-Tolerant Mobile Agent System

Taesoon Park[1], Ilsoo Byun[1], and Heon Y. Yeom[2]

[1] Department of Computer Engineering, Sejong University,
Seoul 143-747, KOREA
{tspark,widepis}@sejong.ac.kr
[2] Department of Computer Science, Seoul National University,
Seoul 151-742, KOREA
yeom@snu.ac.kr

Abstract. In this paper, we propose a low overhead replication scheme for the fault-tolerant mobile agent system. In the proposed lazy replication scheme, execution of a primary agent and migration of its replicas are concurrently processed. Also, the primary agent performs asynchronous consensus with fixed consensus agents so that the consensus step and the replica migration step can concurrently be processed. As a result, the primary agent should not wait for the completion of the replica migration step unless any of the consensus agents fails. The proposed scheme has been implemented on top of the Aglet system and its performance has been measured.

1 Introduction

A mobile agent is a software program which moves from a site to another site to execute a task assigned by a user [1]. As the mobile agent system has drawn attention as a new distributed computing paradigm, the importance of reliable agent execution is more emphasized. Reliable execution of a mobile agent is to guarantee the exactly-once execution of an agent even in case of a system failure [11]. Many fault-tolerance schemes for the mobile agent system have been proposed and they are categorized into the replication schemes [3, 7, 8, 11] and the checkpointing schemes [2, 10, 12].

Replication schemes show a high degree of fault tolerance since a replication scheme with $2k + 1$ replicas can tolerate up to k failures. On the other hand, checkpointing schemes may cause a severe delay in recovery even after a single failure. However, considering the execution time, checkpointing does not require much overhead, while the time to replicate an agent and migrate the replicas and the time to perform the consensus among replicas are not negligible.

To reduce the replication cost, we have suggested asynchronous agent replication schemes and measured the performance [5, 6]: In the asynchronous replication scheme, agent replicas are migrated to the designated sites in an asynchronous manner so that the primary can begin its execution without waiting

for the migration of other replicas. With this optimization, we have achieved up to 37% reduction of the replication cost compared to the synchronous agent replication. However, there is still a performance gap between the systems using the asynchronous replication scheme and no fault-tolerance scheme.

In this paper, to fill the performance gap, we propose a scheme for lazy replication and asynchronous consensus. In the proposed scheme, execution and consensus of an agent may proceed asynchronously with the agent replication. For a new stage, the first replica of a current agent is migrated to the new execution site and begins the execution, while the rest of replicas are sent to the designated sites. Since an agent begins its execution without waiting for the other replicas, it is sometimes possible that the agent ends the stage before its replicas are ready for the consensus. In order to proceed the consensus without waiting for the late replicas, we use fixed consensus agents. Since an agent performs the consensus with fixed consensus agents instead of waiting for the late replicas, there is no delay in consensus unless the primary replica or one of consensus agents fails. To validate the correctness of the proposed scheme and evaluate its performance, we have implemented the lazy replication with asynchronous consensus on top of the Aglet system and measured the performance.

The rest of this paper is organized as follows: Section 2 describes the Aglet system and the failure model. Existing replication and consensus schemes are presented in Section 3 and Section 4 presents the proposed lazy replication scheme with asynchronous consensus. Section 5 describes the experimental environment and discusses the experimental results. Section 6 concludes the paper.

2   The Aglet System

A mobile agent system consists of a number of system sites connected by the communication network. Each of the sites, to support execution and migration of agents, provides one or more *places*. An agent executes its task *on the place* and migrates *between the places*. While residing in a place, the agent performs an assigned task. The execution of an agent in a place and the migration of the agent into the next place are called a *stage*. In other words, the computation of an agent is denoted by a sequence of stages. Figure 1 shows the execution of a mobile agent, MA_i, consisting of four stages. In the figure, $SG_{i,\alpha}$ denotes the α-th stage of MA_i. The task execution and the migration of a stage, $SG_{i,\alpha}$, are denoted by $E_{i,\alpha}$ and $M_{i,\alpha}$, respectively.
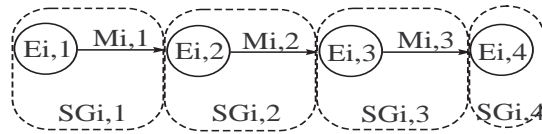

Fig. 1. Stages of a Mobile Agent, MA_i

The *Aglet* [4] is a Java-based mobile agent system. To support execution, migration and communication of agents, the system provides the *AgletContext* environment. The agents in the Aglet system inherit the properties and the methods from the *AgletClass* and perform event-driven activities. For the inter-agent communication, a message-passing mechanism is used in the Aglet system and the *AgletProxy* is provided to support the location transparency of the agent. The AgletProxy is an interface to an Aglet object and every message is sent to the Aglet object through the AgletProxy, regardless of its location.

Failures considered in the system are the *agent failure*, the *place failure* and the *system failure*. For all of these failure types, the fail-stop [9] model is assumed; that is, once a component fails, it stops its execution and does not perform any malicious actions.

3 Replication and Consensus

3.1 Synchronous Replication

For the fault-tolerant execution of an agent, one execution stage consists of three steps, which are the task execution step, the replication step, and the consensus step. Figure 2 shows an example of these three steps where one primary agent and two replicas are used.
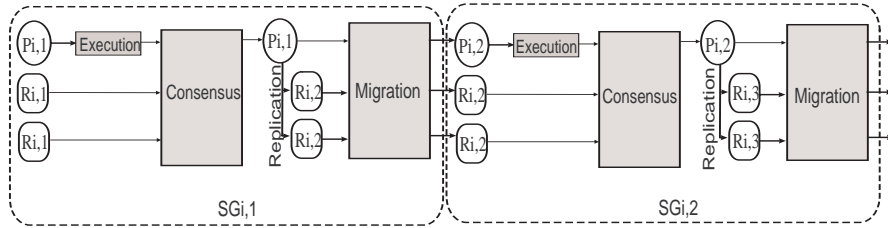

Fig. 2. Synchronous Agent Replication

• Agent Replication: Before a stage, $SG_{i,\alpha}$, begins, the primary agent, $P_{i,\alpha-1}$, of the previous stage, $SG_{i,\alpha-1}$, makes its replicas. A primary agent is the one responsible for the initial task execution of a stage and at the end of the stage, it is also responsible for the agent replication for the next stage. During the replication step, $P_{i,\alpha-1}$ makes $2k$ replicas and migrates them to $2k$ different sites. $P_{i,\alpha-1}$ then migrates itself to the next execution site and becomes a new primary $P_{i,\alpha}$ for the next stage, $SG_{i,\alpha}$. Every replica, $R_{i,\alpha}$, observes the task execution step and the replication step of $P_{i,\alpha}$. In case that a replica suspects the failure of a primary, it may become a new primary after a consensus step.

• Task Execution and Consensus: The primary, $P_{i,\alpha}$, begins the task execution step for the stage, $SG_{i,\alpha}$, as soon as it arrives in a new execution site. When

the primary successfully completes the task, it begins the consensus step. The consensus step is to confirm the task completion of the primary and to prevent any redundant execution by false failure detection of a replica. For example, due to slow execution or communication, a replica may suspect the failure of a primary and try to become a new primary. In such a case, two primary agents may execute the same task, which violates the exactly-once execution property.

The primary begins the consensus by sending out the *consensus_begin* message to every replica, $R_{i,\alpha}$. Every $R_{i,\alpha}$ replies with the *consensus_ack* message unless it has already sent out the message for the same stage. When the primary receives the majority of *consensus_ack* messages, it sends out *consensus_confirm* messages to the replicas and completes the consensus step. If the primary fails to obtain the majority of *consensus_ack* messages, it gives up the current stage and undoes the executed task. In the consensus step, only one primary can obtain the majority votes for a stage and complete the task.

• Failure Handling: A replica $R_{i,\alpha}$ is made to detect any failure of the primary for the stage $SG_{i,\alpha}$. For the failure detection, the time-out is often used. If $R_{i,\alpha}$ cannot receive any *consensus_begin* message within a time-out period, it suspects the failure of the primary during the task execution step. Also, if $R_{i,\alpha}$ cannot receive the majority of the *successful_migration* messages within a time-out period, it suspects the failure of the primary during the replication step. For this, the *successful_migration* of every replica $R_{i,\alpha+1}$ is acknowledged not only to the primary $P_{i,\alpha}$ but also to every $R_{i,\alpha}$. When a failure of the primary is suspected, the replica $R_{i,\alpha}$ becomes a new primary through its own consensus and begins an alternative task execution step for the stage $SG_{i,\alpha}$.

Any replica which first detects the failure of the primary can initiate the consensus step and take over the execution as proposed in [11]. Or as in [8], the priority of every replica is predetermined and only the replica with the highest priority can initiate the consensus step when a primary fails. In this scheme, the $(j+1)$-th replica can take over the task execution step of the current stage, when up to the j-th replica fails. In either case, the agent can keep alive as long as $k+1$ replicas survive failures. Among them, one replica is for the task execution and k replicas are for the majority voting. Therefore, the replication scheme with one primary and $2k$ replicas can tolerate up to k failures.

3.2 Asynchronous Replication

To reduce the replication and migration cost of the synchronous scheme, we have proposed an asynchronous agent replication scheme [5], in which the replica migration proceeds asynchronously with the primary execution.

• Asynchronous Agent Replication: In the replication step, the primary, $P_{i,\alpha}$, makes $2k + 1$ replicas and transfers them as in the synchronous replication scheme. However, every replica in the asynchronous replication scheme has a predetermined priority and the replica with the highest priority becomes the new primary $P_{i,\alpha+1}$ when it arrives in the next site for the stage, $SG_{i,\alpha+1}$. Since the new primary, $P_{i,\alpha+1}$, is the first replica migrated by the previous primary,
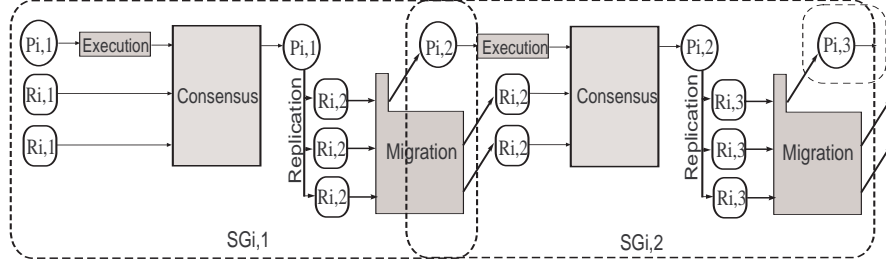
Fig. 3. Asynchronous Agent Replication

$P_{i,\alpha}$, and $P_{i,\alpha}$ may still process the migration of the other replicas, the task execution step of $P_{i,\alpha+1}$ and the replication step by $P_{i,\alpha}$ may proceed concurrently as shown in Figure 3. $P_{i,\alpha}$ terminates itself when it completes the replication step. Because of this asynchronous replication, the time for the replication step by $P_{i,\alpha}$ can fully or partially be masked by the task execution step of $P_{i,\alpha+1}$. As a result, the total execution time of the agent can be reduced.

    • Failure Handling: The failure detection during the task execution step and the replication step is processed in the same way as in the synchronous replication scheme. One difference is that the primary $P_{i,\alpha}$ in the asynchronous scheme may finish the task execution step before some replicas arrive in their execution sites. In this case, $P_{i,\alpha}$ should wait so that every replica can participate in the consensus step. Another problem may happen when $P_{i,\alpha}$ fails before replicating and migrating the majority of the replicas for the next stage. Note that in this case, any replica $R_{i,\alpha}$ becomes a new primary and re-processes the task execution step of the stage $SG_{i,\alpha}$, while the next primary $P_{i,\alpha+1}$ may have already begun its task execution step for the stage $SG_{i,\alpha+1}$. However, even in this case, $P_{i,\alpha+1}$ cannot obtain the majority of votes during the consensus step and therefore it can be discarded anyway.

4   Lazy Replication and Asynchronous Consensus

Asynchronous agent replication can reduce the total execution time of an agent as much as the overlapped time of the replication step of a primary, $P_{i,\alpha}$, and the task execution step of the next primary, $P_{i,\alpha+1}$. However, $P_{i,\alpha+1}$ still has to wait before the consensus step unless the execution step of $P_{i,\alpha+1}$ is longer than the replication step of $P_{i,\alpha}$. In the Aglet system, for a primary agent to communicate with its replicas, it has to obtain the proxy of the replicas first. In the synchronous scheme, the primary migrates with the proxy information of its replicas, since it migrates after all the replicas. However, in the asynchronous scheme, the primary, $P_{i,\alpha}$, should send the proxy information of the replicas, $R_{i,\alpha+1}$s, to the next primary, $P_{i,\alpha+1}$, after it completes the replication step. As a result, $P_{i,\alpha+1}$ cannot begin the consensus step until the replica proxy information arrives.

• Fixed Consensus Agents: To eliminate the waiting time for late replicas, we propose to use fixed consensus agents. The replication scheme with one primary and $2k$ replicas is designed to tolerate up to k failures. This means that only the majority of replicas have a chance for the alternative task execution and the others are made just for attending the consensus step. Assuming the consensus algorithm using the predetermined priority [8], the k replicas with lower priorities are sure to attend only the consensus step. Therefore, in the proposed scheme, we replace the k lower priority replicas with fixed consensus agents.

A consensus agent contains simple codes to perform the consensus step. In the first stage of agent execution, k consensus agents are created and sent to k different sites. Therefore, the primary agent in any stage is sure to know the location of consensus agents and have their proxy information. Now, for the consensus, a primary or any replica suspecting the failure of the primary sends consensus messages to the consensus agents. To differentiate the consensus messages of different stages, consensus related messages should carry the stage number. Consensus agents then reply with the *consensus_ack* or *consensu_nak* messages according to the consensus algorithm.

Using fixed consensus agents, we can take the following advantages: First, a primary makes only $k + 1$ replicas and the time to make k more replicas can be eliminated. Also, a primary no longer waits for the late replicas. As soon as the primary completes the task execution step, it can begin the consensus step with fixed consensus agents. As a result, there is virtually no blocking of primary agent execution when there is no failure.
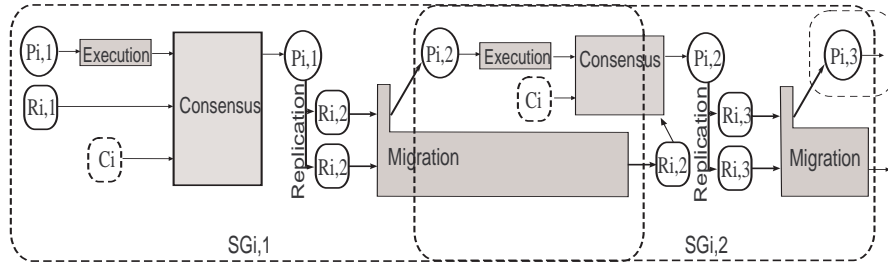

Fig. 4. Lazy Agent Replication and Asynchronous Consensus

• Lazy Replication and Asynchronous Consensus: In the proposed scheme, a primary $P_{i,\alpha}$ performs the asynchronous replication as described in the previous section. Therefore, while $P_{i,\alpha}$ processes the replication step, the next primary, $P_{i,\alpha+1}$, may complete the task execution step for the stage, $SG_{i,\alpha+1}$. When $P_{i,\alpha+1}$ completes the task execution step, it can begin the consensus step by sending out the *consensus_begin* messages to the consensus agents. In case that all the consensus agents are alive, $P_{i,\alpha+1}$ can obtain the majority of *consensus_ack* messages from fixed consensus agents and then proceed the replication step for

the next stage, $SG_{i,\alpha+2}$. Using fixed consensus agents whose location information is known, the primary agent can complete the consensus step without waiting for the migration of all the other replicas.

Figure 4 shows an example of lazy replication and asynchronous consensus with one primary, one replica and one fixed consensus agent, C_i. As it can be noticed from the figure, a primary $P_{i,\alpha+2}$ can begin the task execution step while the previous primaries, $P_{i,\alpha+1}$ and $P_{i,\alpha}$, still perform the replication step. However, for the proposed scheme to be complete, the replication step of each stage should be performed in a very lazy manner. One responsibility of a primary, $P_{i,\alpha}$, is to notify every replica, $R_{i,\alpha}$, of the successful completion of the consensus. Also, each of the next replicas, $R_{i,\alpha+1}$, made by $P_{i,\alpha}$ should know the location information of their previous replicas, $R_{i,\alpha}$s, to inform the $successful\_migration$ message.

To handle these processes without blocking the agent execution, the primary, $P_{i,\alpha}$, proceeds with the replication step and migrates the first replica for the next stage, $SG_{i,\alpha+1}$, after it successfully completes the consensus step with consensus agents. $P_{i,\alpha}$ then waits for the location information of the late replicas of the current stage. The rest of the consensus step and the replication step can be continued when it receives the necessary location information of the other replicas. To complete the consensus step, $P_{i,\alpha}$ sends the $consensus\_confirm$ message to every $R_{i,\alpha}$ instead of $consensus\_begin$ message and continues the migration of the replicas for the next stage. As a result, in the proposed scheme, the consensus step of a stage can be overlapped with the replication step for the next stage and also the replication step of several stages can be overlapped.

• Failure Handling: The role of the replica agent in the lazy replication scheme is the same as in the other replication schemes. A replica $R_{i,\alpha}$ may suspect the failure of its primary if it cannot receive any $consensus\_begin$ or $consensus\_confirm$ message within a time-out period. It also suspects the failure during the replication step, if it cannot receive the majority of $successful\_migration$ messages. In the lazy replication scheme, when a primary, $P_{i,\alpha}$, ends the replication step, it informs the fixed consensus agents of the beginning of the next stage. The location information of current replicas, $R_{i,\alpha}$s, is also carried in that information.

On the receipt of that information, consensus agents act as the replicas of the next stage, $R_{i,\alpha+1}$, and send $successful\_migration$ messages to the previous replicas, $R_{i,\alpha}$. Therefore, even if any new replica may fail on the arrival of a new site, previous replicas can receive the majority of $successful\_migration$ messages as long as more than $k+1$ agents among the primary, replicas and consensus agents survive. The failure of a consensus agent may affect the performance of the agent execution, since a primary cannot asynchronously complete the consensus step if a consensus agent fails. However, in such a case, the primary can wait for the location information of any replica and process the consensus step with that replica. Therefore, even in case of the consensus agent failure, the primary can complete the consensus step, while the execution time may slightly be longer.

5 Performance Study

5.1 Experimental Setup

To validate the correctness of the lazy replication scheme and evaluate its performance, we have implemented the lazy replication and asynchronous consensus scheme (the *LazyRep* scheme) on top of the *Aglet* system. The Aglet system is a Java-based mobile agent system and for our experiments, *Aglet DSK 1.1b2* has been used. The synchronous replication scheme (the *SyncRep* scheme) and the asynchronous replication scheme (the *AsyncRep* scheme) have also been implemented for the performance comparison.

A cluster of five Pentium IV 1 GHz PCs connected by a 100 Mbps Ethernet was used for experiments. Each machine supported one place. An agent traversed the places in a predetermined order and the replicas were also sent to the predetermined places. To obtain a stable performance, we used an agent consisting of twenty stages. For each experimental data, ten runs of the agent execution were measured and then eight measured values were averaged out, excluding the lowest and the highest ones. In each stage, the agent sleeps for T milliseconds, instead of performing any task. To observe the influence of the agent size, an NXN integer array is included in the agent so that we can control the size of the agent by varying the size of the array.

5.2 Experimental Results

Figure 5(a) first compares the agent execution time of three schemes when the replica number is 3, 5 and 7. This number includes the primary, replicas and consensus agents of one stage. The size of the agent is 100 KBytes and the task execution time of one stage is 1000 milliseconds.

Considering the time for the replication step and the consensus step, the SyncRep scheme shows 184% increase as the replica number changes from 3 to 7. Compared to this, the LazyRep scheme shows only 75% increase when the replica number changes. Since the increase of the replication time in the SyncRep scheme is proportional to the number of replicas, the total execution time sharply increases when the replica number is increased. However, the replication time of the LazyRep scheme is not affected by the replica number. Only the consensus time can be longer when the replica number is large, since the primary has to wait for *consensus_ack* messages from more number of consensus agents.

Compared to the LazyRep scheme, the performance of the AsyncRep scheme is somewhat disappointing. When the replica number is three, the AsyncRep scheme achieves 31.6% reduction of the replication and the consensus time, compared to the SyncRep scheme. However, it does not show much reduction when the number of replicas are seven and the performance is even worse for five replicas. One possible explanation is that the stage execution time of 1000 milliseconds may be too short to mask the replication time for a large number of replicas.
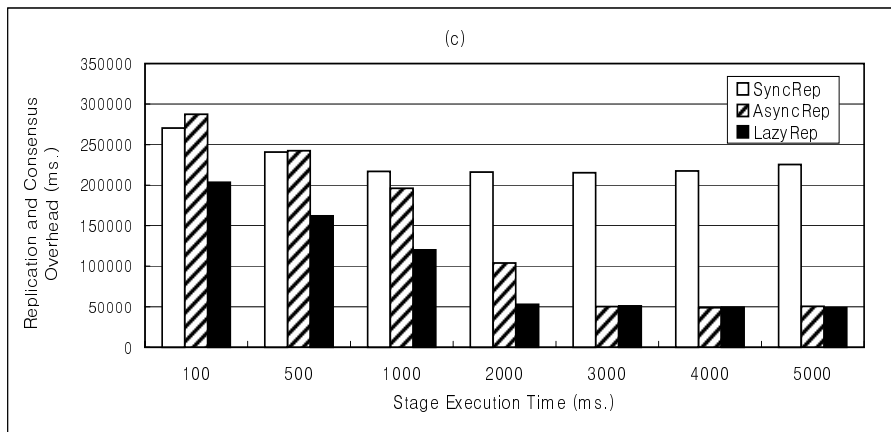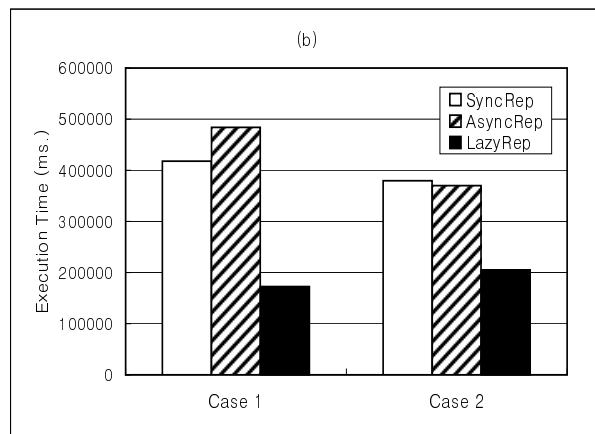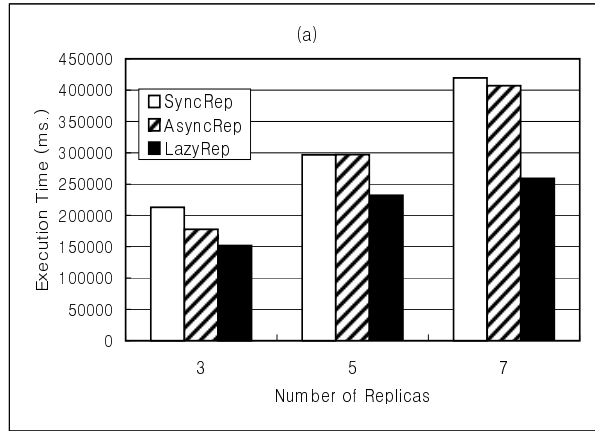
Fig. 5. Experimental Results I

To confirm this, we have measured the performance of two extreme cases as shown in Figure 5.(b). For the Case 1, we have reduced the task execution time of one stage into 100 milliseconds where the agent size is 100 KBytes and the replica number is five. In this case, the AsyncRep scheme is much worse than the SyncRep scheme and this worse performance is due to the time to make replicas. In the AsyncRep scheme, a primary makes $2k+1$ replicas instead of $2k$ replicas. Since copying of one agent takes about 20 milliseconds in this case, the marginal performance gain obtained by asynchronous replication should be lost to copy one more replica.
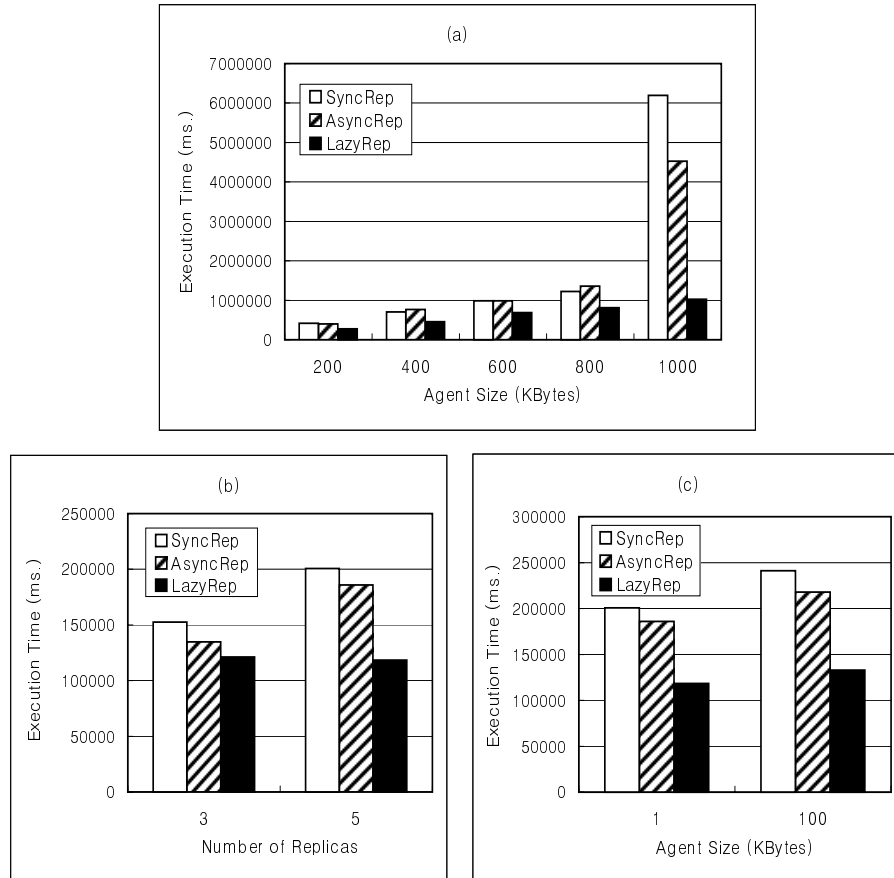

Fig. 6. Experimental Results II

For the performance of the Case 2, we have reduced the agen size to 10 KBytes where the task execution time of one stage is 1000 milliseconds and the replica number is five. By reducing the agent size, the time to make one more replica can

be reduced and also the replica migration time can be reduced. Because of this reduction, the AsyncRep scheme can show the slight performance gain compared to the SyncRep scheme. This disappointing performance of the AsyncRep scheme is only for the cases where the replication time is much longer than the task execution time. When the task execution time becomes long enough to mask the replication step and the consensus step, the performance of the AsyncRep scheme is close to the LazyRep scheme as shown in Figure 5.(c).

The performance of Figure 5.(c) was obtained when the agent size is 100 KBytes and the replica number is five. As shown in the figure, the time for the replication step and the consensus step of the SyncRep scheme is not affected by the task execution time of one stage. However, as the task execution time increases, the replication step and the consensus step of the AsyncRep and the LazyRep schemes can concurrently be processed with the task execution step. As a result, the total execution time of an agent can significantly be reduced.

Figure 6.(a) shows the execution time of three schemes, when the agent size varies from 200 KBytes to 1000 KBytes. The replica number is five and the task execution time of one stage is 1000 milliseconds for this result. A large agent requires more time to make replicas and to migrate them. Therefore, the performance of the SyncRep scheme is heavily influenced by the agent size. The performance of the AsyncRep scheme is also affected a lot by the agent size, however, nonnegligible performance gain can be achieved with asynchronous replication. Compared to this, the LazyRep scheme shows very desirable performance and also very stable performance.

Figure 6.(b) and 6.(c) show the impact of failures on the agent execution time of three schemes. For these results, it is assumed that any agent, including the primary, replicas and consensus agents, can fail on each stage with the probability of 0.1 and the time-out to detect a failure is three seconds. As shown in Figure 6.(b), the LazyRep scheme now achieves only 30%–54.6% reduction of the migration time under the various number of replicas. It also achieves 54.6%–56% reduction of the replication time under the various agent size as shown in Figure 6.(c). Even though re-execution caused by a failure partially takes off the benefit of the LazyRep scheme, Figure 6.(b) and 6.(c) show that the LazyRep scheme is still under the less influence.

6 Conclusions

In this paper, we have proposed a lazy replication and asynchronous consensus scheme for the fault tolerant mobile agent system. In the proposed scheme, the replication step, the consensus step and the task execution step of an agent proceeds asynchronously. To process the consensus step without waiting for the late replicas, we have introduced fixed consensus agents and an agent can perform the consensus step with fixed consensus agents. As a result, there is practically no delay in the consensus step unless the primary or one of consensus agents fails. To evaluate the performance of the proposed scheme, we have implemented the proposed scheme on top of the Aglet system. The performance results show that

the lazy replication with asynchronous consensus scheme can achieve much reduction of the replication and the consensus cost, compared to earlier replication schemes. Also, the lazy replication scheme shows very stable performance.

Acknowledgments

References

1. Baumann, J., Hohl, F., Rothermel, K., Strasser, M.: Mole - Concepts of a Mobile Agent System. World Wide Web Journal, Vol. 1, No. 3 (1998) 12–137
2. Gendelman, E., Bic, L.F., Dillencourt, M.B.: An Application-Transparent, Platform-Independent Approach to Rollback-recovery for Mobile Agent Systems. Proc. of the 20th Int'l Conf. on Distributed Computing Systems (2000)
3. Johansen, D., Marzullo, K., Schneider, F.B., Jacobsen, K.: NAP: Practical Fault-Tolerance for Itinerant Computations. Proc. of the 10th Int'l Conf. on Distributed Computing Systems (1999)
4. Karjoth, G., Lange, D.B., Oshima, M.: A Security Model for Aglets. IEEE Internet Computing (1997)
5. Park, T., Byun, I., Kim, H., Yeom, H.Y.: The Performance of Checkpointing and Replication Schemes for Fault Tolerant Mobile Agent Systems. Proc. of the 21st Symp. on Reliable Distributed Systems (2002) 256–261
6. Park, T., Byun, I.: Low Overhead Agent Replication for the Reliable Mobile Agent System. Lecture Notes in Computer Science, Vol. 2790. Springer-Verlag, Berlin Heidelberg New York (2003) 1170–1179
7. Pleisch, S., Schiper, A.: Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems. Proc. of the 19th Symp. on Reliable Distributed Systems (2000) 11-20
8. Pleisch, S., Schiper, A.: FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach. Proc. of the Int'l Conf. on Dependable Systems and Networks (2001) 215–224
9. Schlichting, R.D., Schneider, F.B.: Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. ACM Transactions on Computer Systems, Vol. 1, No. 3 (1983) 222–238
10. Silva, L., Batista, V., Silva, J.G.: Fault-Tolerant Execution of Mobile Agents. Proc. of the Int'l Conf. on Dependable Systems and Networks (2000)
11. Strasser, M., Rothermel, K.: Reliability Concepts for Mobile Agents. International Journal of Cooperative Information Systems, Vol. 7, No. 4 (1998) 355-382
12. Strasser, M., Rothermel, K.: System Mechanism for Partial Rollback of Mobile Agent Execution. Proc. of the 20th Int'l Conf. on Distributed Computing Systems (2000)