Vol. 44 No. 3 IPSJ Journal Mar. 2003

Regular Paper

Mobile Agent Model for Fault-tolerant Objects Systems

Takao Komiya,† Tomoya Enokido† and Makoto Takizawa†

There are many discussions on how to make servers take checkpoints of servers fault-tolerant, i.e., replication and checkpointing. Applications are also required to be fault-tolerant in order to realize fault-tolerant systems. In this paper, we take a mobile agent approach to realizing fault-tolerant applications on multiple servers. Agents move around servers whose objects are manipulated. In traditional systems, application programs do not work if application servers are faulty. If the server is faulty, the agent finds another server where the agent can be performed. In addition, agents are replicated. Replicas of agents can move to servers even if a replica of the agent is faulty. Thus, applications can be fault-tolerant.

1. Introduction

Systems are composed of object servers and applications. Applications issue requests to object servers to manipulate objects and then the object servers send responses to the applications. There are many discussions on how to make object servers fault-tolerant, i.e. replications [7),11)] and checkpointing protocols [6)]. Even if object servers are fault-tolerant, the system is not operational if applications are faulty. In this paper, we discuss how to realize fault-tolerant applications by using mobile agents [1)]. An agent first lands at an object server and then is performed to manipulate objects in the object server. If the agent finishes manipulating the objects, the agent moves to another server which has objects to be manipulated. Thus, an agent moves around object servers. Here, agents manipulate objects only in the object servers. In addition, an agent negotiates with other agents manipulating objects in a conflicting manner. Through the negotiation, each agent autonomously makes a decision on whether the agent still holds the objects or releases the objects. Thus mobile agents have following characteristics:

1. Agents are autonomously initiated and performed.
2. Agents negotiate with other agents.
3. Agents are moving around server computers.

In order to realize a fault-tolerant system, not only object servers but also applications should be fault-tolerant. The two-phase commitment protocol [5),10)] and protocols for replicating object servers [7),11)] are not robust for faults of application servers while robust for servers' faults. For example, servers might block if applications are faulty in the two-phase commitment protocol [10)]. A mobile agent can move to another object server if one server is faulty. Thus, mobile agents can be still operational as long as at least one object server where the agents can be performed is operational. Agents can be divided into subagents which are performed in parallel. In addition, agents can be replicated and each replica agent is independently performed. Even if one replica agent is faulty, objects can be manipulated through other replica agents. If an agent leaves an object server after manipulating objects, the agent releases the objects. If an agent releases objects before committing or aborting, the agent cannot be aborted, i.e., *unrecoverable*. In order to overcome the difficulty, a *surrogate* agent for an agent is created and is left on the object server when the agent leaves the server. The surrogate agent holds the objects until the agent commits or aborts. In addition, a surrogate agent recreates an agent if the agent on another server is faulty. Then, the agent restarts. There are types of conditions which each agent can commit like atomic and *at-least-one* conditions. An agent negotiates with its surrogate agents left on the servers to check if some condition given for the agent is satisfied. For examples, an agent which atomically manipulates multiple servers commits if all the surrogates are operational. In an at-least-one condition, the agent can commit if at least one surrogate is operational, i.e. an agent updates at least one server. We discuss how to manipulate multiple object servers by using agents in presence of server and application faults.

† Department of Computers and Systems Engineering, Tokyo Denki University

In section 2, we present a system model. In section 3, we present a fault-tolerant agent model. In section 4, we discuss implementation of mobile agents.

2. System Model

2.1 Object Servers

A system is composed of object servers D_1, ..., D_m ($m \geq 1$), which are interconnected with reliable, high-speed communication networks. Each object server supports a collection of objects and methods for manipulating the objects. Objects are encapsulations of data and methods. Objects are manipulated only through methods supported by the objects.

Suppose a pair of subtransactions T_1 and T_2 manipulate an object o in an object server D_i by using methods op_1 and op_2, respectively. Here, if the result obtained by performing the methods op_1 and op_2 on the object o depends on a computation order of op_1 and op_2, the methods op_1 and op_2 are referred to as *conflict* with one another on the object o [5]. A pair of methods op_1 and op_2 are *compatible* if op_1 and op_2 do not conflict. For example, a pair of methods *increment* and *decrement* do not conflict, i.e., are *compatible* on a *counter* object. On the other hand, a method *reset* conflicts with a method *increment* and *decrement* on the *counter* object. If a method op_1 from a transaction T_1 is performed before a method op_2 from another transaction T_2 and the methods op_1 and op_2 conflict, every method $op_1^{'}$ from T_1 is required to be performed before every method $op_2^{'}$ from the transaction T_2 conflicting with the method $op_1^{'}$. This is a *serializability* property of transaction [5]. The locking protocol and timestamp ordering protocol [5] are used to realize the serializability of multiple transactions. In the locking protocol, if one transaction holds an object, then other transactions are required to wait. Transactions issue lock requests to lock an object in an arbitrary order. Deadlock may occur. On the other hand, transactions are totally ordered in their timestamps. Since the objects are held by the transactions according to the timestamp order, there occurs no deadlock.

2.2 Mobile Agents

An agent is a program which can be autonomously performed on one or more than one object server. An agent issues methods to manipulate objects in an object server where the agent exists. Every object server is assumed to support a platform to perform agents. A pair of agents A_1 and A_2 are referred to as *conflict* if the agents A_1 and A_2 manipulate a same object through conflicting methods. Suppose, an agent A_1 issues an *increment* method to an *account* object while another agent A_2 issues a *reset* method. A pair of the methods *increment* and *reset* conflict. Hence, the agents A_1 and A_2 conflict.

First, an agent A is autonomously initiated on an object server. The agent A is first stored in the memory of an object server D_i. If enough resource like memory is allocated for the agent A on the object server D_i, the agent A moves to the object server D_i, i.e., *lands* at D_i. Here, D_i is referred to as *current* object server of the agent A. After landing at an object server D_j, the agent A is allowed to be performed if there is no agent on the server D_j which conflicts with the agent A.

Suppose an agent A manipulates multiple object servers D_1, ..., D_m. There are two ways to manipulate the object servers, *serial* and *parallel* ways. In the serial way, the agent A visits one object server at a time. For example, an agent A first visits an object server D_1, next D_2, ..., and lastly D_m. Here, object server D_{i-1} is referred to as *parent* of the server D_i. An optimal sequence of object servers in which the agent A to visit has to be obtained, e.g., to minimize the computation time and communication time (**Fig. 1**). This shows the serial computation of the agent A.

In the parallel way, an agent A is divided into multiple subagents A_1, ..., A_m. Here, the subagents A_1, ..., A_m are allowed to be independently performed. Each subagent A_i concurrently moves to an object server D_i ($i = 1$, ..., m). After all the subagents finish, the agents are merged into an agent A again (**Fig. 4**). This shows the parallel computation of the agent A.

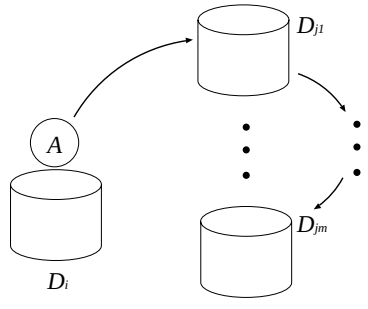

Fig. 1 Optimal routing.

2.3 Termination Conditions

Suppose an agent A manipulates objects in multiple object servers $D_1, \ldots, D_m$ $(m > 1)$. The agent A visits these object servers in serial or parallel ways as shown in **Figs.** 1 and **2**. After finishing manipulating objects in all the object servers, the agent A commits if some *termination* condition C on the object servers $D_1, \ldots, D_m$ is satisfied. Otherwise, the agent A aborts. For example, an agent commits if all the object servers are successfully manipulated. Unless at least one object server is successfully manipulated, the agent aborts, i.e. no update is done on objects in any object server. The two-phase commitment (2PC) protocol is used to realize the atomicity principle in distributed database systems [5]. In another example, an application would like to book one hotel. Suppose there are a pair of hotel object servers H_1 and H_2. An agent A is separated to a pair of subagents A_1 and A_2. The subagents A_1 and A_2 are issued to the object servers H_1 and H_2, respectively, in a parallel way. Each subagent tries to book a hotel. Suppose one subagent A_1 makes a success at booking a hotel but another agent A_2 fails. Since the application would like to book one hotel, the agent A can commit although the agent A does not successfully manipulate one object server H_2. In a serial way, the agent A visits one hotel server, say H_1. If the agent A could book the hotel in the server H_1, the agent A can commit. Otherwise, the agent A moves to another hotel server H_2. Thus, an agent commits if some conditions are satisfied after visiting some number of object servers.

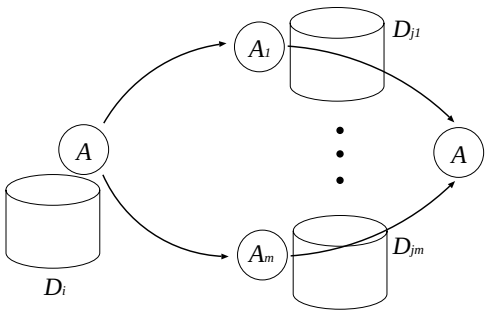

Fig. 2 Split and merge of agents.

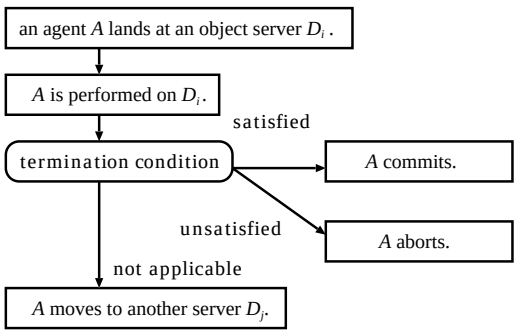

Fig. 3 Behaviour of agent.

There are following types of termination conditions:

[Termination conditions]

1. *Atomic* condition: an agent is successfully performed on all the object servers. If the agent fails on at least one server, the agent aborts. This is an all-or-nothing condition used in the traditional two-phase commitment protocol.
2. *Majority* condition: an agent is successfully performed on more than half of the object servers. If at least half of the servers are not successfully manipulated, the agent aborts.
3. *At-least-one* condition: an agent is successfully performed on at least one object server. If the agent is not successfully performed on all the object servers, the agent aborts.
4. $\binom{n}{r}$ condition: an agent is successfully performed on more than r object servers $(r \leq n)$. If the agent is not successfully performed on at least r servers, the agent aborts. □

More general conditions are discussed in a paper [9]. Each agent A is assumed to have a termination condition $Term(A)$ given by an application. The agent A commits if $Term(A)$ is satisfied after manipulating some number of object servers. Otherwise, the agent A aborts. If $Term(A)$ is not applicable, the agent A moves to another server (**Fig. 3**).

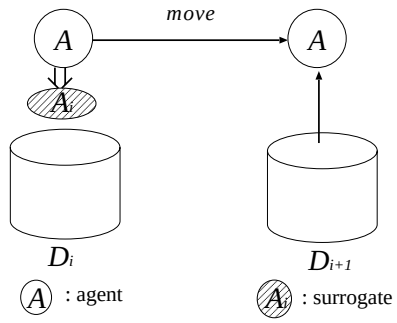

Fig. 4 Surrogate agent.

3. Fault-Tolerant Agents

3.1 Surrogates

There are two types of faults, faults of object servers and agents to occur in a system. First, object servers may be faulty due to crash. If an object server to which an agent would like to move is faulty, the agent has to find another candidate object server. For example, if an object server is replicated, another replica is found. Next, agents may be faulty as well. If an object server where an agent exists is faulty, the agent is also faulty. Here, the agent is aborted.

Suppose an agent A finishes on an object server D_i. Here, if the agent A leaves the object server D_i, objects manipulated by the agent A in the server D_i are released and can be used by other agents. After visiting other servers, the agent A cannot abort because the agent A already committed on the server D_i, i.e., the agent is *unrecoverable*. In addition, if an object server where an agent A exists is faulty, the agent A is also faulty. Here, the agent A cannot be recovered because the agent A crashes. In order to resolve these problems, an agent A creates a *surrogate* agent A_i of the agent A on an object server D_i before the agent A leaves the server D_i (**Fig. 5**). The agent A is referred to as *parent* of each surrogate agent A_i. A surrogate agent A_i of an agent A behaves as follows:

[**Behavior of surrogate agent**]

1. A surrogate agent A_i is created for an agent A in an object server when the agent A leaves the object server.
2. The surrogate agent A_i holds objects manipulated by the parent agent A in the object server until the agent A terminates. The surrogate A_i does not move to another object.
3. On arrival of another agent B, the surrogate A_i negotiates with the agent B if B conflicts with A_i.
4. The surrogate A_i also negotiates with the parent agent A and the other surrogates of A to make a decision on commit or abort.
5. The surrogate A_i recreates an agent A if the agent A is faulty. □

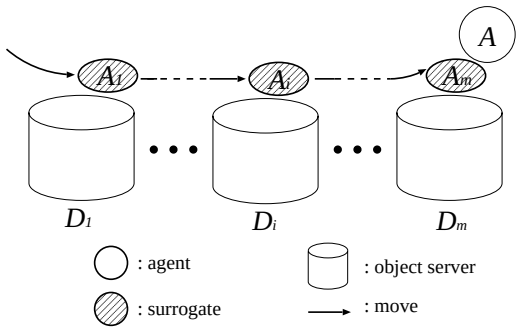

Fig. 5 Surrogate agents.

As shown in Figure 4, suppose a surrogate agent A_i is created on an object server D_i after a surrogate A_{i-1} on another object server D_{i-1}. Here, the surrogate agent $A_j (j < i)$ is referred to as *preceding* surrogate of the surrogate agent $A_i (A_j \rightarrow A_i)$. A_{i-1} is the *directly preceding* surrogate of $A_i (A_j \Rightarrow A_i)$. On the other hand, $A_j (j > 1)$ is a *succeeding* subagent of the surrogate agent A_i. A_{i+1} is the *directly succeeding* surrogate of the surrogate agent A_i. In Figure 5, $A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_m$. A pair of surrogates A_i and A_j of an agent A are *concurrent* iff neither $A_i \rightarrow A_j$ nor $A_j \rightarrow A_i$.

Suppose another agent B might come to an object server D_j after the agent A leaves the object server D_j. Here, the agent B negotiates with the surrogate agent A_i if B conflicts with the agent A_i. Depending on the negotiation, the agent B might take over the surrogate A_i. Thus, when the agent A finishes visiting all the object servers, some preceding surrogate A_i of the agent A may not exist. The agent A starts the negotiation with its surrogates $A_1, \ldots, A_m$. If a termination condition $Term(\mathrm{A})$ on the surrogates $A_1, \ldots, A_m$ is satisfied, the agent A commits. For example, an agent commits if all the surrogates safely exist in the atomic condition. If one surrogate had aborted, the agent aborts. If the agent terminates, i.e. commits or aborts, the surrogates of the agent A are annihilated. Here, other agents conflicting with the agent A are allowed to manipulate objects which are released by A_i.

3.2 Agent Fault

Agents and surrogate agents may be faulty. For example, an agent is faulty if the current object server of the agent is faulty. Suppose an agent A moves to an object server D_j from an object server D_i. A surrogate A_i of the parent agent A is left on the object server D_i. Suppose the server D_j is faulty after the agent A lands at the object server D_j. Here, the agent A is also faulty. The preceding surrogate A_i communicates with the agent A. If the surrogate A_i could not communicate with the agent A, A_i finds that A is faulty. Here, the sur-

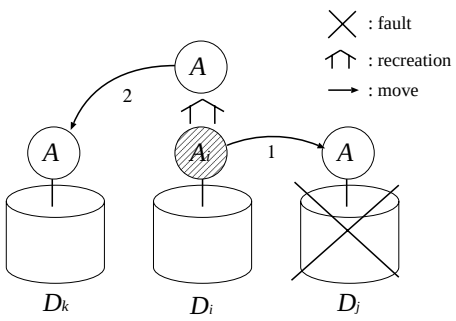

Fig. 6 Recreations of agent.

rogate A_i recreates an agent A on the server D_i and then the agent A finds another operational server D_k for which the agent A to leave (**Fig. 6**). That is, the agent A rolls back to the previous state shown by the surrogate A_i and then restarts. Thus, a surrogate agent can play a role of checkpoint of an agent.

Surrogates may be also faulty. In Figure 4, suppose that a surrogate agent A_i is faulty due to the fault of an object server D_i while an agent A exists on an object server D_m. It depends on a termination condition on surrogates how a faulty surrogate recovers. For example, there is no need to recover the faulty surrogate A_i if the at-least-one condition is taken. The agent A could not commit without the surrogate A_i in the atomic condition. Here, A_i is required to be recovered. One way to recover a surrogate A_i is that a directly preceding surrogate A_{i-1} recreates a surrogate agent A' and issues the agent A' to another object server D'_i where the agent A' can be performed, e.g. a replica of the server D_i. After the agent A' is performed on the server D'_i, a surrogate A'_i is left and A' is annihilated. Here, the new surrogate A'_i takes over the faulty surrogate A_i. That is, A'_i is a directly succeeding surrogate of A_{i-1} and a directly preceding surrogate of A_{i+1} i.e. $A_{i-1} \Rightarrow A'_i \Rightarrow A_{i+1}$.

Suppose an agent A is in parallel performed by subagents $A_1, \ldots, A_m$ as shown in Figure 3. Here, suppose a subagent A_i is faulty. As stated here, there is no need to recover the faulty subagent A_i if the agent A could commit without A_i, e.g. at-least-one condition is taken. Otherwise, a subagent A_i is recreated by a directly preceding surrogate of A_i.

Let A denote an agent or a surrogate which is faulty. A surrogate A_i preceding the faulty surrogate A ($A_i \rightarrow A$) detects the fault of A. If A is an agent, a surrogate A_i directly preceding the agent A ($A_i \Rightarrow A$) recreates a new instance of the agent A. There might be more than one surrogate directly preceding the faulty surrogate agent A. In this case, the surrogates directly preceding the surrogate A negotiate with each other and one surrogate is taken to recreate a surrogate A. For example, a surrogate whose identifier is minimum is taken.

Next, suppose A is a surrogate agent. Let A_i be a surrogate directly preceding the surrogate agent A ($A_i \Rightarrow A$). There is a surrogate A_j which is preceded by the surrogate A ($A \Rightarrow A_j$). There are two cases with respect to whether or not the surrogate A is required to be performed according to the termination condition. Here, A_i has to know which surrogates the surrogate A precedes if A is required to be performed. Each surrogate A_i has information on which surrogates precedes the surrogate A_i and where the agent is. First, the surrogate A_i communicates with the agent to get information of surrogates preceded by the faulty surrogate A. Then, the surrogate A_i recreates an agent and finds another server where the agent can be performed. If found, the agent is performed on the server. On completion, a surrogate is left but the agent is annihilated as presented preceded in this subsection. If the agent is faulty, the surrogate A_i cannot get information on the preceding surrogates of the faulty surrogate A. Here, the surrogate A_i has to wait for the recovery of the agent. In order to overcome this difficulty, the surrogate A_i periodically communicates with the agent and obtains the information on preceding surrogates of the agent. If the surrogate A_i had the information from the agent, the surrogate A_i detects a surrogate A_j preceded by the faulty surrogate A from the information. However, the surrogate A_j may now not be a surrogate preceded by the faulty surrogate A due to some fault. The surrogate A_i first communicates with the surrogate A_j and checks whether or not A_j is still the surrogate. If so, A_i recreates an agent to recover the computation done by the faulty surrogate A. If A_j is not the surrogate, the surrogate A_i has to wait for the recovery of the agent.

3.3 Deadlock

Suppose an agent A_1 passes over an object server D_1 and is moving to another server D_2, and another agent A_2 passes over D_2 and is moving to D_1 as shown in **Fig. 7**. If a pair of the agents A_1 and A_2 conflict on each of the servers D_1 and D_2, neither A_1 can be performed

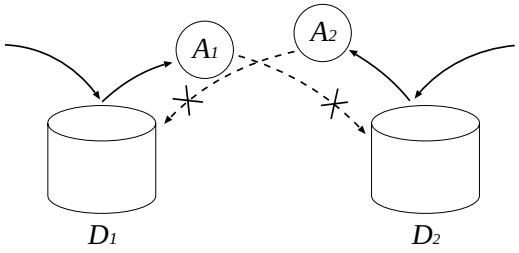

Fig. 7 Deadlock.

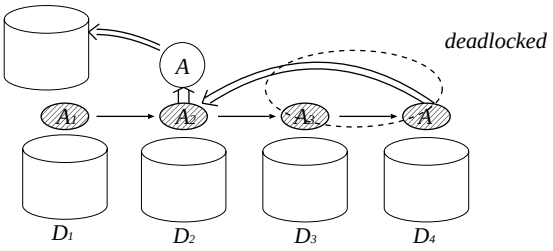

Fig. 8 Retreat.

on D_2 nor A_2 can be performed on D_1. Here, deadlock occurs.

Suppose an agent A is on an object server D_4 after visiting object servers D_1, D_2, and D_3, and the agent A cannot be performed because A is deadlocked. One way to resolve the deadlock is that the agent A is aborted. In stead of aborting all the computation done by the agent A, only a part of the computation required to resolve the deadlock is tried to be aborted. Suppose a surrogate A_3 is also included in a same deadlock cycle as the agent. The surrogate A_3 recreates an agent A. Since the agent A is still deadlocked, the surrogate A_3 is also aborted. The agent A is referred to as *retreated* to a surrogate A_2 (**Fig. 8**). The surrogate A_2 recreates an agent A on the server D_2 and then the agent A finds another server D_5 on which the agent A can be performed.

4. Implementation

An agent is implemented in Java [2),8)] and Aglets [1)]. Oracle8i database systems [4)] on Windows2000 are used as object servers which are interconnected in 100base Ethernet. An agent manipulates table objects in Oracle object servers by issuing SQL commands, select, update, insert, and delete.

As presented before, after an agent leaves an object server, a surrogate of the agent stays on the object server while the surrogate agent holds objects manipulated by the agent. The surrogate agent releases the object only if the agent commits or aborts. In this implementation, an agent A and its surrogates are realized as follows (**Fig. 9**). Here, suppose an agent A lands at an object sever D_i.

1. An agent A manipulates objects in an object server D_i by issuing SQL [3)] commands.
2. A *clone* A' of the agent A is created if the agent A finishes manipulating objects in the object server D_i. The clone A' leaves the server D_i for another server D_j. Here, the clone A' is an agent A.
3. The agent A stays on the object server D_i as a surrogate.

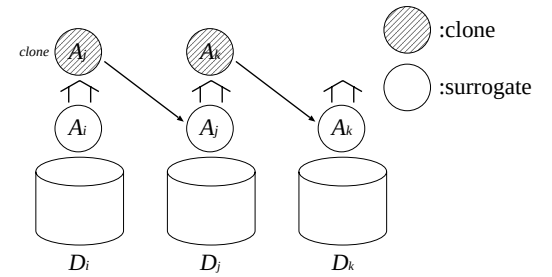

Fig. 9 Surrogates.

Thus, a clone of an agent A is created and moves to another server as an agent. The agent A is just performed on the object server D_i and then is changed to the surrogate. If the agent A leaves the object server D_i, locks on objects held by the agent are released. Therefore, an agent A stays on an object server D_i without releasing the objects. A clone of the agent leaves the object server D_i for another object server D_j. Here, the clone of the agent A plays a role of the agent A in the server D_j.

If all the object servers are manipulated, an agent makes a decision on commit or aborts by communicating with the surrogates as discussed in this paper. If *commit* is decided, every surrogate A_i commits on an object server D_i.

Suppose an agent B comes to an object server D_i. If the agent B conflicts with the agent A, the agent B negotiates with the surrogate A_i on D_i. If B takes over A_i through the negotiation, the surrogate A_i is aborted.

5. Concluding Remarks

This paper discussed a mobile agent model for processing fault-tolerant transactions which manipulate multiple object servers. There are many discussions on how to make servers fault-

tolerant. However, there is not enough discussion on how to make application fault-tolerant. In this paper, applications are realized by mobile agents. An agent first moves to an object server and then manipulates objects. The agent autonomously moves around the object servers to perform the computation. If the agent conflicts with other agents in an object server, the agent negotiates with the other agents. After leaving an object server, a surrogate of an agent is left on the server. If the agent A is faulty on a server, the surrogates on servers which A visited recreate the agent A. In addition, an agent is replicated and the replicas are performed in parallel. In the mobile agent model, we can increase reliability and availability since agents do not suffer from faults.

References

1) Aglets Software Development Kit Home. http://www.trl.ibm.com/aglets/.
2) The Source for Java (TM) Technology. http://java.sun.com/.
3) Database Language SQL, American National Standards Institute (1986). Document ANSI X3.135.
4) Oracle8i Concepts Vol. 1, Oracle Corporation (1999). Release 8.1.5.
5) Bernstein, P., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison Wesley (1987).
6) Chandy, K.M. and Lamport, L.: Distributed Snapshots : Determining Global States of Distributed Systems, *ACM TOCS*, Vol.3, No.1, pp.63–75 (1985).
7) Garcia-Molina, H. and Barbara, D.: How to Assign Votes in a Distributed System, *ACM*, Vol.32, No.4, pp.841–860 (1985).
8) Omicini, A., Zambonelli, F., Klusch, M. and Tolksdorf, R.: *Coordination of Internet Agents* (2001).
9) Shimojo, I., Tachikawa, T. and Takizawa, M.: M-ary Commitment Protocol with Partially Ordered Domain, *Proc. 8th Int'l Conf. on Database and Expert Systems Applications(DEXA'97)*, pp.397–408 (1997).
10) Skeen, D.: Nonblocking Commitment Protocols, *Proc. ACM SIGMOD*, pp.133–147 (1982).
11) Wiesmann, M. et al.: Understanding Replication in Databases and Distributed Systems, *Proc. IEEE ICDCS-2000*, pp.264–274 (2000).

(Received July 3, 2002)
(Accepted October 7, 2002)

Takao Komiya was born in 1978. He received his B.E. degree in Computers and Systems Engineering from Tokyo Denki University, Japan in 2001. He is now a graduate student of the master course in the Deptartment of Computers and Systems Engineering, Tokyo Denki University. His research interests include distributed systems and agent system. He is a student member of IPSJ.

Tomoya Enokido was born in 1974. B.E. and M.E. degrees in Computers and Systems Engineering from Tokyo Denki University, Japan 1997 and 1999. After he worked for NTT Data Corporation, he is currently a research assistant in the Department of Computers and Systems Engineering, Tokyo Denki University. His research interests include distributed systems and group communication. He is a member of IPSJ.

Makoto Takizawa served as program co-chair of the IEEE International Conference on Distributed Computing Systems (ICDCS) in 1998 and as program vice chairs of ICDCS in 1994 and 2000, and is serving as a general co-chair of ICDCS-2002. He also served as a general co-chair of IEEE ISORC. He is a member of the program committees of many IEEE Computer Society conferences including ICDCS, SRDS, ICPADS, ISORC, and ICNP. He was elected for 2003–2005 BoG member of IEEE Computer Society. Takizawa is a full professor in the Department of Computers and Systems Engineering, Tokyo Denki University, Japan. He is now a dean of the graduate school of Science and Engineering, Tokyo Denki University. He chaired the Information Division at the Research Institute for Technology, Tokyo Denki University from 1998 to 2002. He was a visiting professor at GMD-IPSI, Germany (1989–1990) and has been a regular visiting professor at Keele University, England since 1990. Takizawa is a fellow of Information Processing Society of Japan (IPSJ) and was a member of the executive board of IPSJ from 1998 to 2000. He chaired SIGDPS (distributed processing) of IPSJ from 1997 to 2000 and was an editor of the Journals of IPSJ (1994–1998). Takizawa received his BE, ME, and DE in computer science from Tohoku University, Japan. In 1996, he won the best paper award at IEEE International Conference on Parallel and Distributed Systems (ICPADS). He is a member of the IEEE and a member of the ACM and IPSJ.