Mobile Agent Model for Fault-Tolerant Objects Systems

Takao Komiya, Tomoya Enokido, and Makoto Takizawa
Tokyo Denki University, Japan
{komi, eno, taki}@takilab.k.dendai.ac.jp

Application programs may be faulty as well as server systems. There are many discusses on how to make servers fault-tolerant, i.e. replication and checkpointing. Applications are also made fault-tolerant by in order to realize fault-tolerant systems. In this paper, we take an agent approach where the applications are realized by mobile agents. Agents move around object 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, replicas of agents move to operational servers even if some replicas suffer from faults. In the mobile agent approach, applications can be fault-tolerant.

耐障害オブジェクトシステムのためのモバイルエージェントモデル

小宮 貴雄　榎戸 智也　滝沢 誠
東京電機大学理工学部

アプリケーションプログラムはサーバと同様に障害被ることがある。耐障害サーバを構築するための方法について議論されている。アプリケーションもまた、耐障害システムを実現するために耐障害に作成されなければならない。本論文は高信頼アプリケーションを実現するために、モバイルエージェントを使用する。エージェントと呼ばれるプログラムはオブジェクトサーバを移動する。従来のシステムでは、アプリケーションサーバが障害しているならば、アプリケーションプログラムは動作しない。本論文の提案するエージェントでは、アプリケーションサーバが障害したならば、そのエージェントはもう一つのエージェントが実行可能なサーバを見つける。さらに、エージェントのレプリカはいくつかのレプリカが障害にあったとしても運用可能なサーバに移動する。モバイルエージェントアプローチにおいて、アプリケーションプログラムは耐障害となり得る。

1 Introduction

Systems are composed of object servers and applications. Applications issue requests to object servers and then the object servers send response to the applications. There are many discussions on how to make object servers fault-tolerant, i.e. replications [7, 17] and checkpointing protocols [5]. Even if object servers are fault-tolerant, the system is not operational if applications are faulty. In this paper, applications are realized as 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. An agent moves around object servers. Here, agents manipulate objects only in local object servers. In addition, an agent negotiates with the agent if some agents manipulate objects in a conflicting manner. Through the negotiation, each agent autonomously makes a decision on whether the agent continues to hold 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 computers.

The two-phase commitment protocol [4, 14] and protocols for replicating object servers [7, 17]are not robust for faults of application servers while robust for servers' faults. Mobile agents 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. 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. In order to overcome the difficulty, a *surrogate* agent for the agent is created and is left on the object server. The surrogate agent holds the objects until the agent commits or aborts. We discuss how to manipulate multiple object servers by using agents in presence of server and application faults.

In section 2, we present a system model. In section 3, we present a fault-tolerant agent model. In section 4, we discuss how to resolve confliction among agents. In section 5, we discuss implementation of mobile agents.

2 System Model

2.1 Object servers

A system is composed of object servers $D_1, \ldots, D_m$ ($m \geq 1$), 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 in an object server D_i by using methods op_1 and op_2, respectively. Here, if the result obtained by performing op_1 and op_2 depends on a computation order of op_1 and op_2, op_1 and op_2 are referred to as *conflict* with one another on the object [4]. For example, *read* and *write* conflict on a *file* object. A pair of methods *increment* and *decrement* do not conflict, i.e. are *compatible* on a *counter* object. On the other hand, *reset* conflicts with *increment* and *decrement* on the *counter* object. If a method from a transaction T_1 is performed before a method from another transaction T_2 and the methods conflict, every method op_1 from T_1 is required to be performed before every method op_2 from T_2 conflicting with the method op_1. This is a *serializability* property of transaction [4]. The locking protocol and timestamp ordering protocol [4] are used to realize the serializability. In the locking protocol, if one transaction holds an object, then other transactions are regard to wait. Transactions lock an object in an arbitrary order. On the other hand, transactions are totally ordered in their timestamps. The objects are held by the transactions according to the timestamp order.

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.

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 a *current* object server of the agent A.

Suppose an agent A lands at a server D_i to manipulate an *account* object through a method *increment*. Here, suppose another agent B is now *resetting* the *account* object. Since *reset* conflicts with *increment*, the agent A cannot start. A pair of agents A_1 and A_2 are referred to as *conflict* if A_1 and A_2 manipulate a same object through conflicting methods. After landing at an object server D_j, the agent A is allowed to be performed if there is no agent on D_j which conflicts with A.

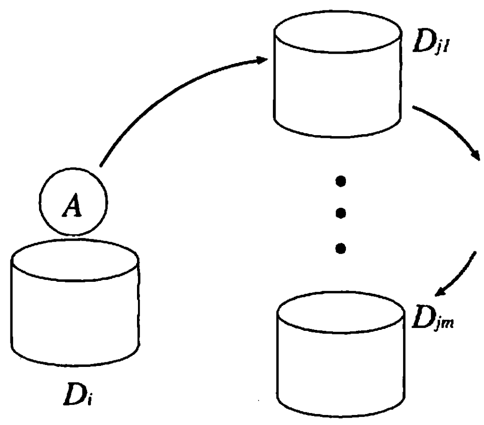

Figure 1: Optimal routing.

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 discussed before. After finishing manipulating objects in all the object servers, the agent A commits if some *consensus* condition C on the object servers D_1, ..., D_m is satisfied. Otherwise, the agent A aborts. For example, an agent commits if all the object servers are successfully manipulated. Otherwise, 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 [4]. In another example, an application would like to book one hotel. Suppose there are a pair of hotel object server H_1 and H_2. An agent A is separated to a pair of subagents A_1 and A_2. A_1 and A_2 are issued to the object servers H_1 and H_2, respectively. Each subagent tries to book a hotel. Suppose one subagent A_1 makes a success at booking a hotel but A_2 fails. Since the application would like to book one hotel, the agent A can commit although the agent A successfully manipulates only one object server. Thus, if at least one of the object servers is successfully manipulated, the agent A commits. There are following types of consensus conditions:

[Consensus conditions]

1. *Atomic consensus*: an agent is successfully performed on all the object servers. This is a all-or-nothing principle consensus condition used in the traditional two-phase commitment protocol.
2. *Majority consensus*: an agent is successfully performed on more than half of the object servers.
3. *At-least-one consensus*: an agent is successfully performed on at least one object server.
4. $\binom{n}{r}$ *consensus*: an agent is successfully performed on more than r object servers ($r \leq n$). □

More general consensus conditions are discussed in a paper [13]. Each agent A is assumed to have a consensus condition *Cons*(A) given by an application. The agent A commits if *Cons*(A) is satisfied after manipulating object servers. Otherwise, the agent A aborts.

3 Fault-Tolerant Agents

3.1 Surrogates

There are two types of faults, object servers and agents faults to occur in a system. First, object servers may be faulty, i.e. crash. If an object server

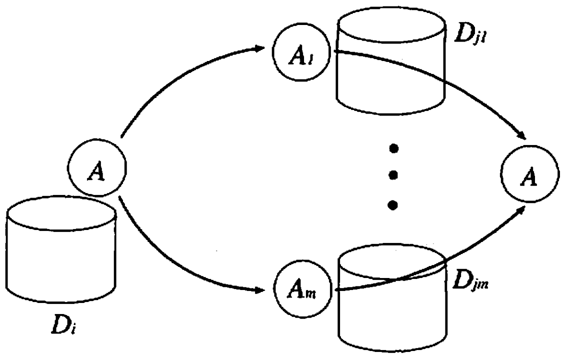

Figure 2: Split and merge of agents.

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 is faulty, the agent is also faulty. Here, the agent is aborted.

Suppose an agent A finishes on a server D_i. Here, if the agent A leaves the object server D_i, objects manipulated by the agent A in 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 D_i [Figure 4].

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, the agent A first visits the object server D_1, next D_2, ..., and lastly D_m. 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 [Figure 1]. In the parallel way, the agent A is divided into multiple subagents A_1, ..., A_m. Each subagent A_i concurrently moves to an object server D_i. Here, the subagents A_1, ..., A_m are required to be independently performed. After all the subagents finish, the agents are merged into an agent A again [Figure 3]. The agent A is referred to as *parent* of each surrogate agent A_i. The surrogate agent A_i plays following roles :

1. The surrogate agent A_i holds objects manipulated by the agent A until the agent A termi-

nates. The surrogate A_i does not move to another object.

2. The surrogate A_i negotiates with other agents conflicting with A_i.
3. 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.
4. The surrogate A_i recreates an agent A if A is faulty.

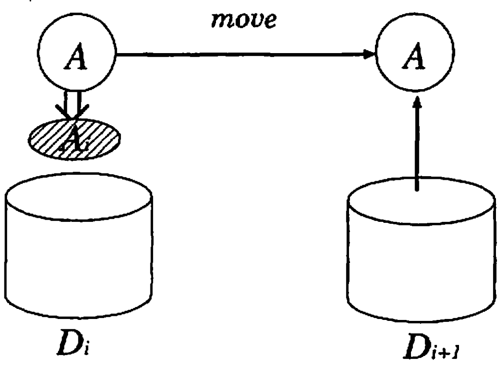

Figure 3: Surrogate agent.

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 D_{i-1}. Here, $A_j (j < i)$ is referred to as *preceding* surrogate of A_i. A_{i-1} is the *most preceding* surrogate of A_i. On the other hand, $A_j (j > 1)$ is a *succeeding* subagent of A_i. A_{i+1} is the *most succeeding* surrogate of 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 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 surrogate A_i of A may not exist. The agent A starts the negotiation with its surrogates A_1, ..., A_m. If a consensus condition C 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 consensus 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.

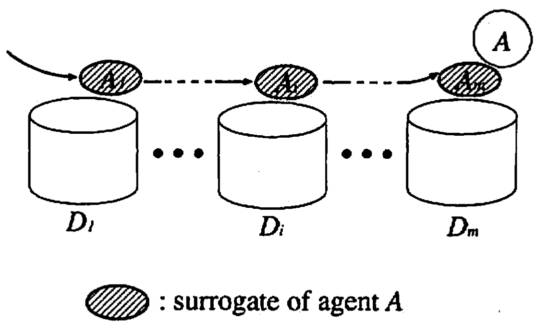

Figure 4: Surrogate agents.

3.2 Agent fault

Agents and surrogate agents are faulty if object servers where the agents exist are 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 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 surrogate A_i recreates an agent A on D_i and then the agent A finds another operational server D_k for which A to leave [Figure 5]. That is, the agent A rolls back to the previous state shown by the surrogate A_i and then restarts.

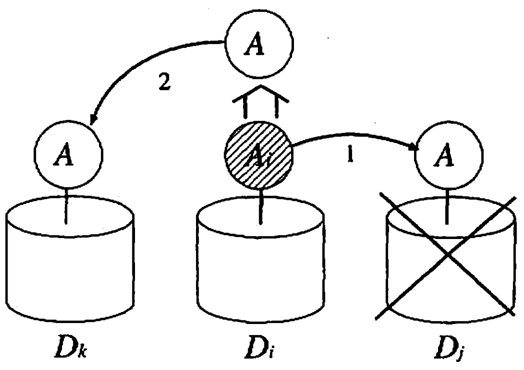

Figure 5: Recreations of agent.

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 consensus condition on surrogates how a faulty surrogate recovers. For example, there is no need to recover the faulty surrogate A_i if at-least-one consensus condition is taken. If the agent A could not commit without the surrogate A_i like atomic consensus condition, A_i is required to be recovered. One way to recover a surrogate A_i is that a preceding surrogate A_{i-1} recreates a surrogate agent A' and issues the agent A' to another object server D'_i where 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 most succeeding surrogate of A_{i-1} and a most preceding surrogate of A_{i+1}.

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 consensus condition is taken. Otherwise, a subagent A_i is recreated by a most preceding surrogate of A_i.

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 Figure 6. If a pair of the agents A_1 and A_2 conflict on each of D_1 and D_2, neither A_1 can be performed on D_2 nor A_2 can be performed on D_1. Here, deadlock occurs.

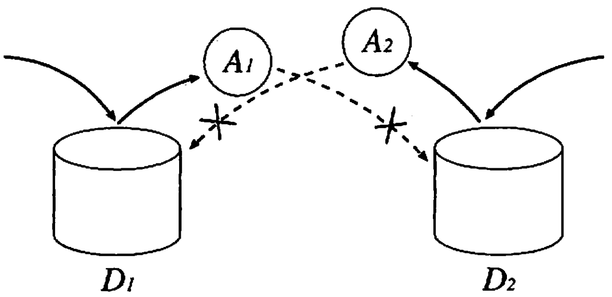

Figure 6: Deadlock.

Suppose that 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. 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 A can be performed.

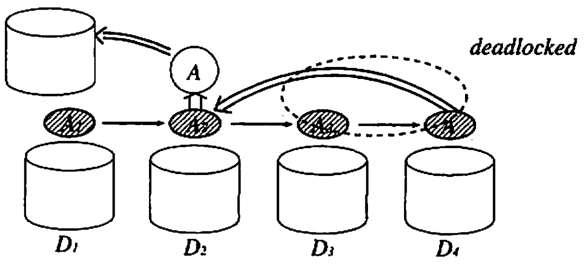

Figure 7: Retreat.

4 Implementation

An agent is implemented in Java [11,16] and Aglets [1]. Oracle8i database systems [12] 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 [Figure 8]. 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 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.

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 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 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 abort 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.

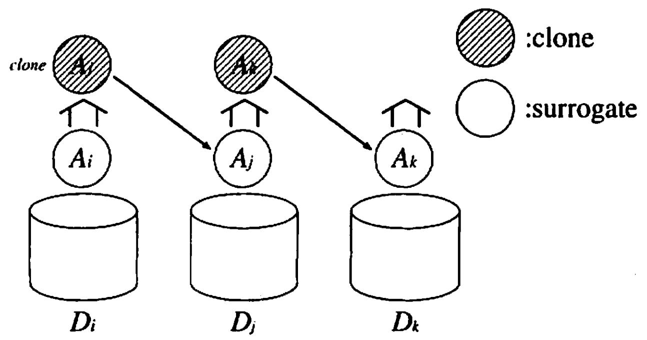

Figure 8: Surrogates.

If the agent B conflicts with the agent A, B negotiates with the surrogate A_i on D_i. If B takes over A_i by 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. 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] American National Standards Institute, "Database Language SQL," *Document ANSI X3.135*, 1986,

[3] Barrett, P. A., Hilborne, A. M., Bond, P. G., and Seaton, D. T., "The Delta-4 Extra Performance Architecture," *Proc. 20th Int'l Symp. on FTCS*, 1990, pp. 481–488.

[4] Bernstein, P.A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison Wesley*, 1987.

[5] Chandy, K. M. and Lamport, L., "Distributed Snapshots : Determining Global States of Distributed Systems," *ACM TOCS*, Vol. 3, No. 1, pp. 63–75, 1985.

[6] Date, C. J., "Introduction to Database System," *Prentice-Hall*, 1994.

[7] Garcia-Molina, H. and Barbara, D., "How to Assign Votes in a Distributed System," *Journal of ACM*, Vol.32, No.4, 1985, pp.841-860.

[8] Mattern, F., "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms* (Cosnard, M. and Quinton, P. eds.), *North-Holland*, Amsterdam, 1989, pp.215-226.

[9] Mehdi, J. and Wolfgang, L. "A Component-based Mobile Agent System," 1999.

[10] Nagi, K., "Transactional Agents : Towards a Robust Multi-Agent System," *LNCS* No. 2245, *Springer-Verlag*, 2001.

[11] Omicini, A., Zambonelli, F., Klusch, M. and Tolksdorf, R., "Coordination of Internet Agents," *Springer-Verlag*, 2001.

[12] Oracle Corporation,"Oracle8*i* Concepts Vol. 1," Release 8.1.5, 1999.

[13] Shimojo, I., Tachikawa, T., and Takizawa, M., "M-ary Commitment Protocol with Partially Ordered Domain," *Proc. of the 8th Int'l Conf. on Database and Expert Systems Applications (DEXA'97)*, 1997, pp.397-408.

[14] Skeen, D., "Nonblocking Commitment Protocols," *Proc. of ACM SIGMOD*, 1982, pp.133-147.

[15] Tanaka, K. and Takizawa, M., "Quorum-based Locking Protocol in Nested Invocations of Methods," Proc. of *DEXA'2001*, 2001, pp.857-866.

[16] The Source for Java (TM) Technology, http://java.sun.com/.

[17] Wiesmann, M., et al., "Understanding Replication in Databases and Distributed Systems," *Proc. of IEEE ICDCS-2000*, 2000, pp.264-274.