マルチメディア通信と分散処理 111－30
コンピュータセキュリティ 20－30
（２００３．２．２８）

Design and Implementation of Transactional Agents

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

Mobile agents move around object servers where the agents manipulate objects. A transactional agent is an agent which manipulate objects in one or more than one object server so as to satisfy some constraint. There are some types of constraints depending on applications. ACID is an example of the constraints, which shows traditional transactions. There are other constraints like at-least-one constraint when a transaction can commit if at-least-one object server is successfully manipulated. We discuss how transactions with types of constraints can commit. We discuss how to implement transactional agents.

トランザクションエージェントの設計と実装

白石 雅　小宮 貴雄　榎戸 智也　滝沢 誠
東京電機大学理工学部情報システム工学科

本論文では、エージェントにより、複数のデータベースサーバを操作する問題を論じる。エージェントは ACID 等のコミットメント制約のもとで、複数のデータベースサーバを操作する。エージェントは、データベースサーバ内のオブジェクトを操作し、完了したならば次のサーバに移動する。この論文では、エージェントにより、種々のコミットメント条件を満たしたなら、複数のデータベースサーバを操作する方法について論じている。又、実装について論じる。

1. Introduction

In traditional client-server applications, application programs on clients or a application servers issue requests to object servers like database servers. Application programs and objects exist in clients and servers, respectively. On the other hand, any computers can have programs and objects in peer-to-peer (P2P) applications. In the P2P applications, huge number of computers are interconnected in the network and the computers are not so reliable as server computers. Hence, connections with mobile stations are often disconnected. Applications cannot manipulate objects in servers due to the disconnection.

In database applications, transactions manipulate objects so as to satisfy ACID (atomicity, consistency, isolation, and durability) properties [6]. For example, objects in multiple object servers are required to be atomically manipulated. In the traditional systems, objects are locked to serialize multiple transactions [6, 8, 10]. In timestamp ordering protocol [6], transactions are totally ordered in their timestamps. Transactions manipulate objects according to the timestamp order, i.e. the elder, the earlier. In addition to supporting the serializability, the atomic manipulation of multiple servers has to be supported. The two-phase commitment protocol [6,10] is widely used to realize the atomicity among multiple database systems. The two-phase commitment protocol supports robustness against server faults but not against application fault, i.e. servers may block due to client faults [15].

In another computation paradigm, programs named *mobile agents* [1] manipulate objects by moving around object servers. An agent first lands at an object server and then is performed to manipulate objects in the object server. Agents manipulate objects only in local object servers without issuing requests to remote object servers in a network. After manipulating all or some object servers, an agent makes a decision on commit or abort. For example, an agent commits only if all the object servers are successfully manipulated. Thus, each agent has its own commitment condition. In addition, an agent negotiates with another agent if the agent manipulates objects in a conflicting manner. Through the negotiation, each agent autonomously makes a decision on whether the agent continues to hold the objects or gives up to hold the objects. We discuss how transactional agents manipulate multiple object servers by using agents in presence of server and application faults.

In section 2, we present a model of object server. In section 3, we present an agent model for processing transactions which manipulate multiple object servers. In section 4, we discuss how agents negotiate with other agents. In section 5, we discuss commitment conditions of transactional agents. In section 6, 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. The networks are assumed to be reliable, i.e. messages are delivered to destinations in sending order with neither duplication nor loss of message. Each object server supports a collection of objects and methods for manipulating the objects. Objects are encapsulations of data and methods.

Each object server supports following methods to manipulate objects in the server:

1. $begin\text{-}trans$: A subtransaction starts. Methods issued by the subtransaction are recorded in the log.
2. $op(o)$: A method op is performed on an object o.
3. $prepare$: The log of a subtransaction is saved in a stable memory.
4. $commit$: A database is physically updated by using the log and a subtransaction commits.

5. $abort$: A subtransaction aborts, i.e. database is not updated and log is removed.

If result obtained by performing a pair of methods 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 $conflict$ on the object. For example, a pair of methods *increment* and *decrement* do not conflict, i.e. are $compatible$ on the $counter$ object. On the other hand, $reset$ conflicts with *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_3 from T_1 is required to be performed before every method op_4 from T_2 conflicting with the method op_3. This is a $serializability$ property of transaction [6, 8]. There are locking protocols [6, 8, 10] and timestamp ordering protocol [6] to realize the serializability.

If a transaction manipulates objects in multiple object servers, the two-phase commitment protocol [6] is used to realize the atomic manipulation of the objects in the object servers. The commitment protocol is robust for failure of object server. However, if the application server is faulty, all the operational object servers might block [6, 15]. Thus, the two-phase commitment protocol is not robust against client fault.

3. Computation Model of Agent

An agent is a program which can be autonomously performed on one or more than one object server. An agent issues methods to an object server to manipulate objects in an object server where the agent exists. For example, a procedure of an agent is written in Java [3, 13]. Every object server is assumed to support a platform to perform agents.

First, an agent A is autonomously initiated on an object server. The procedure and data of an agent A are first stored in the memory of an object server D_i. If enough resource like memory to perform the agent A is allocated for the agent A on the object server D_i, the agent A can move to the object server D_i, i.e. the agent A can $land$ at the object server D_i. Here, the object server D_i is referred to as $current$ for the agent A.

Suppose an agent A lands at an object server D_i to manipulate an *account* object through a method $increment$. Here, suppose another agent B is not *resetting* the *account* object. Since $reset$ conflicts with *increment*, the agent A cannot be started. 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. After landing at an object server D_j, the agent A is allowed to be performed on the object server D_j if there is no agent on an object server D_j which conflicts with an agent A.

Suppose an agent A is at an object server D_i. After finishing manipulating the object, the agent A moves to another agent D_j [Figure 1]. Suppose there are multiple possible object servers $D_{j1}, \ldots, D_{jm}$ $(m > 1)$ where the agent A can land. Let $Cand_i(A)$ be a $candidate$ server set, i.e. a collection of the possible object servers $\{D_{j1}, \ldots, D_{jm}\}$ at which an agent A can land from an object server D_i. For example, there are replicas $D_{j1}, \ldots, D_{jm}$ of some object server D_j.

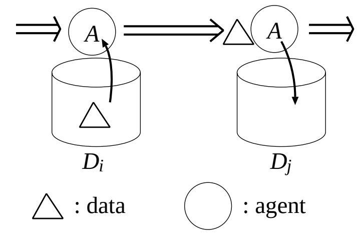

Figure 1. Agent.

$Cand_i(A)$ is a cluster $C(D_j)$ of the replicas $D_{j1}, \ldots, D_{jm}$. For example, if an agent A only reads objects, one replica server D_{jk} is selected and then moves to the object server D_{jk}. If the agent A updates objects, all the object servers in $C(D_j)$ are manipulated by the agent A. This is similar to a famous two-phase locking (2PL) protocol [6]. On the other hand, an agent A issuing a *read* method visits object servers in a subset Q_r. The candidate set $Cand_i(A)$ is a $read$ quorum. The agent A issues $write$ method to object servers in a $write$ quorum Q_w. The agent A visits all the object servers in Q_w. Here, $Q_r \cap Q_w \neq \phi$ and $Q_r \cup Q_w = Cand_i(A)$. If A conflicts with other agents on a replica, A waits. This shows a quorum-based protocol [7].

An agent A can be replicated in $A_1, \ldots, A_m$ $(m \geq 2)$. Each replica A_i is autonously performed. By replicating an agent, parallel processing and fault- tolerance can be realized.

4. Model of Transactional Agent

4.1. Commitment conditions

An agent A manipulates objects in multiple object servers by moving around the object servers. A scope $Scp(A)$ of an agent A means a set of object servers which A possibly manipulate. For example, an agent manipulate replicas of object servers. Here, the scope of the agent is a set of the replicas. If an agent A finishes manipulating each object server D_i, the commitment condition $Com(A)$ of the agent A is checked. For example, an agent A commits if all the servers are successfully manipulated.

[Commitment conditions]

1. $Atomic\ commitment$: an agent is successfully performed on all the object servers, i.e. all-or-nothing principle. This is a commitment condition used in the traditional commitment protocols [8, 15].
2. $Majority\ commitment$: an agent is successfully performed on more than half of the object servers.
3. *At-least-one* $commitment$: an agent is successfully performed on at least one object server.
4. $\binom{n}{r}$ $commitment$: an agent is successfully performed on more than r out of n object servers $(r \leq n)$.
5. $General\ commitment$: some condition is satisfied for the object servers. $\square$

The atomic, majority, and at-least-one commitment conditions are shown in forms of $\binom{n}{n}$, $\binom{n}{\ulcorner (n+1)/2 \urcorner}$, and $\binom{n}{1}$ commitment conditions, respectively. More general commitment conditions with preference are discussed in a paper [14].

Each agent A is assumed to have a commitment condition $Com(A)$ given by an application. There are still discussions on when the commitment condition $Com(A)$ of an agent A can be applied while the agent A is moving an object server. Let $H(A)$ be a set of object servers, possibly ordered, which an agent A has manipulated, i.e. passed over($H(A) \leq Scp(A)$). In the atomic commitment condition, $Com(A)$ can hold only of all the object servers to be manipulated are manipulated, i.e. $H(A) = Scp(A)$. On the other hand, $Com(A)$ can hold over if only one object server is manipulated, i.e. $H(A) = 1$ in the at-least-one commitment condition.

If an agent A leaves an object server D_i, an agent named $surrogate$ of A is left on D_i [Figure 2]. The surrogate agent A_i still holds objects in the object server D_i manipulated by the agent A on behalf of the agent A.

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 of the agent A if the agent B conflicts with the agent A. After 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 may not exist, due to the fault and negotiation with other agents. The agent A starts the negotiation procedure with its surrogates $A_1, \ldots, A_m$. If a commitment condition $Com(A)$ 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 commitment condition. As discussed in the following section, surrogates do negotiation with other agents. Then, the surrogate may abort if another agent is decided to take over objects held by the surrogate by the negotiation. If the surrogates exist, the computation performed by the agent can be successfully terminated. Then, the surrogate agents of the agent A are annihilated. Here, other agents conflicting with the agent A are allowed to manipulate objects.

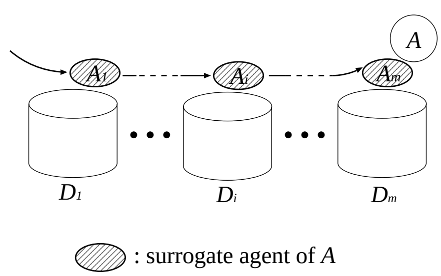

Figure 2. Surrogate agents.

As discussed here, a surrogate may be aborted in the negotiation with other agents or due to the fault of the object server. There are two states of each surrogate B_j, *abortable* and *commitable*. If the surrogate B_j is in $abortable$ state, B_j can be aborted. For example, if another agent A conflicting with the surrogate B_j takes over the surrogate B_j through the negotiation between A and B_j, the surrogate B_j aborts. The agent B of the surrogate B_j eventually tries to commit. The agent B informs all the surrogates of $commit$ by sending $Prepare$ messages. On receipt of the $prepare$ message, the surrogate B_i enters $commitable$ state where update data is saved in a log. Here, the surrogate B_j does not abort in the negotiation.

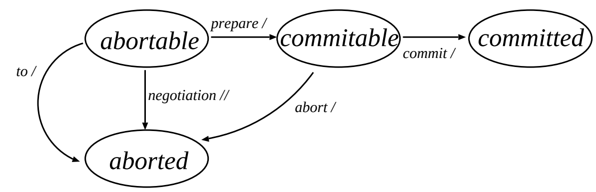

Figure 3. States of surrogate.

4.2. Resolution of confliction

Suppose an agent A moves to an object server D_j from another object server D_i. An agent A cannot be performed on an object server D_j if there is an agent or surrogate B conflicting with A. Here, the agent A can take one of the following ways:

1. The agent A in D_i waits until the agent A can land at an object server D_j.
2. The agent A finds another object server D_k which has objects to be possibly manipulated before the object server D_j.
3. The agent A negotiates with the agent B in the object server D_j.
4. The agent A $aborts$.

Suppose there are other agents $B_1, \ldots, B_k$ which are being performed on the object server D_j. Each agent B_i shows an agent or surrogate agent of an agent. If the agent A conflicts with some agent B_j on an object o, the agent A negotiates with the agent B_j with respect to which agent A or B_j holds the object o [Figure 4]. There are following negotiation policies:

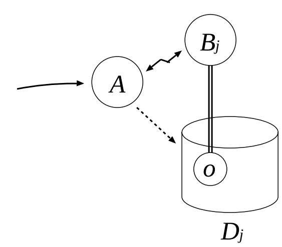

Figure 4. Conflicting agents.

[Negothiation policies]

1. The agent A blocks until the agent B_j commits.
2. The agent A takes over the agent B_j. That is, the agent B_j releases the objects and blocks, and then the agent A starts.
3. The agent B_j aborts and the agent A starts. □

The first way is similar to the locking protocol. An agent A blocks if some agent B holds an object o in a conflicting way with the agent A. If the agent B waits for release of an object held by the agent A, a pair of the agents A and B are deadlocked. Thus, deadlock among agents may occur. When an agent A blocks in an object server D_i, a timer is started. If the timer expires, the agent A takes one of the following ways:

1. The agent A retreats to an object server D_j which A has passed over. The surrogates of the agent A which

have been performed before the object server D_j are aborted. Then, the surrogate A_j on D_j restarts.

2. Every surrogate A_j of the agent A initiates a deadlock detection agent $LD_j(A)$.

In the second way, an agent A takes over an agent B_j in an object server D_j if the agent B holds an object and the agent A conflicts with B_j. Here, the agent A starts the negotiation with the agent B_j on the object server D_j by using a following negotiation protocol :

[Negotiation protocol]

1. An agent A sends a $can\text{-}I\text{-}use$ message $CIU(o, op)$ to an agent B_j on an object server D_j. This means that an agent A would like to manipulate an object o through a method op in an object server D_j.
2. On receipt of a message $CIU(o, op)$ from an agent A, an agent B_j sends an OK message to the agent A if the agent B_j can release the object o or the agent B_j does not mind if the agent A manipulates the object o. Here, there are two approaches to the agent B_j's releasing the object o :
   a. The agent B_j aborts if the agent A precedes the agent B_j, e.g. the priority of the agent A is higher than the agent B.
   b. The agent B_j rolls back to a checkpoint and then restarts if the agent A precedes the agent B_j.

   Otherwise, the agent B_j sends a No message to the agent A.
3. On receipt of OK from the agent B_j, the agent A starts manipulating the object o.
4. On receipt of No from the agent B_j, there are following ways:
   a. The agent A blocks until the agent A receives OK/NO from the agent B_j.
   b. The agent A aborts. □

If the agent B_j agrees with the agent A in the negotiation protocol, the agent A can manipulate objects by taking over the agent B_j. In the second way, the agent B_j not only releases the object but also aborts. Each agent autonomously makes a decision on which way to be taken through negotiation with other conflicting agents.

4.3. Decisions

There are two types of agents, ordered agents and unordered agents. Every pair of ordered agents manipulate objects in a well-defined way. Each ordered agent A is assigned a $precedent$ identifier $pid(A)$. An agent A_1 $precedes$ another agent A_2 ($A_1 \rightarrow A_2$) iff $pid(A_1) < pid(A_2)$. For example, a *timestamp* [6] can be used as an identifier of an agent. That is, the identifier $pid(A)$ of an agent A is time $ts(A)$ when the agent A is initiated at the home server. An agent A_1 precedes another agent A_2 only if $ts(A_1) < ts(A_2)$. If the timestamp with identifier of home server is used as a precedent identifier of an agent, either A_1 precedes A_2 or A_2 precedes A_1 for every pair of different agents A_1 and A_2. That is, the agents are totally ordered in the precedent identifiers. If a logical clock like vector clock [12] is used as precedent identifier, the agents are partially ordered in the precedent identifiers. An agent A_1 is concurrent with another agent A_2 ($A_1 \,||\, A_2$) iff neither A_1 precedes A_2 nor A_2 precedes A_1. Here, the agents A_1 and A_2 can be performed on an object server in any order.

Suppose multiple agents $A_1, \ldots, A_m (m>1)$ would like to manipulate an object o in an object server D_i and the agents conflict with each other. The agents $A_1, \ldots, A_m$ are ordered by using the precedent identifiers of the agents. Suppose $pid(A_1) < \ldots < pid(A_m)$. An agent A_s manipulates an object o before another agent A_t if $pid(A_s) < pid(A_t)$. If a pair of the agents A_s and A_t are concurrent ($A_s \,||\, A_t$), the agents A_s and A_t are allowed to be performed on the object o in any order. However, if a pair of the agents A_s and A_t conflict on a pair of object servers D_i and D_j, the agents A_s and A_t are required to be performed in a same order at the object servers D_i and D_j. There never occurs deadlock.

Like locking protocols, an unordered agent can obtain an object if no conflicting agent obtains the object. 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 the object server D_2 and is moving to D_1 as shown in Figure 5. If a pair of the agents A_1 and A_2 conflict on each of the object servers D_1 and D_2, neither the agent A_1 can be performed on the object server D_2 nor the agent A_2 can be performed on the object server D_1. Here, deadlock occurs.

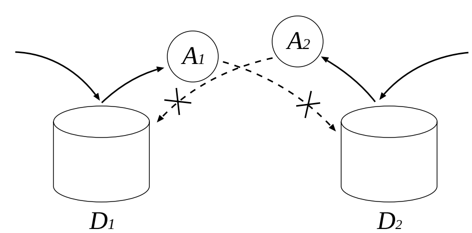

Figure 5. Deadlock.

Here, an agent B_j means an "agent" or a surrogate agent in the object server D_j. An agent A would like to be performed on an object server D_j but conflicts with an agent B_j in D_j. First, suppose an agent B_j is a surrogate of an agent B. The surrogate agent B_j makes a following decision depending on the commitment conditions:

1. The surrogate B_j takes the at-least-one commitment principle: If the surrogate B_j knows at least one surrogate of the agent B exists, the surrogate B_j releases the object and aborts. The surrogate B_j informs the other surrogates of this abort.
2. The surrogate B_j takes the majority commitment principles: If the surrogate B_j knows more than half of the surrogates of B exist, the surrogate B_j releases the object and aborts. The surrogate B_j informs the other surrogates of this abort.
3. The surrogate B_j takes the $\binom{n}{r}$ commitment: If the surrogate B_j knows more than r surrogate agents of the agent B exist, the surrogate B_j releases the object and aborts. □

5. Implementation

5.1. Environment

An agent is implemented in a pair of ways Aglets [1] and Telescript [16]. Relational database systems Sybase [4] and Oracle8i [5] on Solaris, Linux, and Windows2000 are used as object servers which are interconnected in a100base Ethernet. Each object server supports an XA interface [11] for the two-phase commitment.

An agent manipulates table objects in object servers by issuing SQL [9] commands, select and update. A mobile agent realized in Telescript can carry the state to other object servers i.e. process of agent is migrated. However, Aglets agent cannot bring the state to other object servers, just text and heap area are transferred.

An object server is realized in an Oracle and Sybase object server. JDBC(Java database connectivity) [2] is used to realize a program interface to an object server. The JDBC class is required to be loaded to an Aglet agent in order for the agent to issue SQLs on an object server. A $home$ computer of an agent means a computer where the agent is initiated. In order to perform an agent on an object, JDBC is required to exist on the home computer or the server. If JDBC does not exist on the server, JDBC on the home computer is transfered to the server. It takes about 10 sec. to transfer and load JDBC on 100-base LAN. In the Internet, it takes about 34 sec. to transfer the JDBC class between Saitama and Kanagawa. Some object server may not support JDBC. Each type of object server, i.e. Oracle and Sybase, requires an agent to use its own type of JDBC. Hence, an agent cannot move to an object server if the object server does not support its JDBC and the home computer does not other. Next, suppose the home computer supports JDBC. An agent moves to one of object servers D_1 and D_2. Here, D_1 has JDBC but D_1 does not. If the agent moves to D_2, it takes a large time than D_1. Thus, it is an important decision factor of a route whether an object server support JDBC or not.

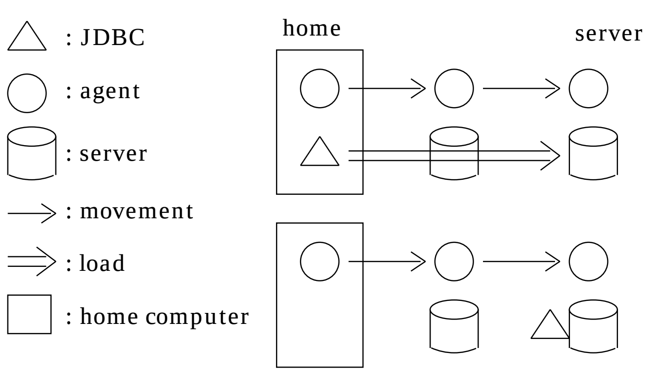

Figure 6. Agent on JDBC.

5.2. Surrogates

As presented before, after an agent leaves an object server, a surrogate agent of the agent stays on the object server while the surrogate agent holds objects manipulated by the agent. The surrogate agent releases the object on time when the agent commits or aborts. In this implementation, an agent and its surrogates are realized as follows [Figure 7]. Here, suppose an agent lands at an object sever D_i by using SQL with some consistency.

1. An agent A manipulates objects in an object server D_i.
2. A clone A' of the agent A is created if the agent A finishes manipulating objects in an object server D_i. The clone A' leaves the object server D_i for another object server D_j.

Thus, a clone of an agent A is created and moves to another object server as an agent. The agent A is just performed on the object server D_i and then is changed to the surrogate. If an agent leaves the object server D_i, locks on objects held by the agent are released. Therefore, an agent stays on an object server D_i and a clone of the agent leaves the object server D_i for another object server D_j.

If all the object servers required by the commitment condition are successfully manipulated, an agent makes a decision on commit or abort by communicating with the surrogates as discussed in this paper. If *commit* is decided by the commitment condition, a surrogate commits on an object server D_i. Otherwise, a surogate aborts.

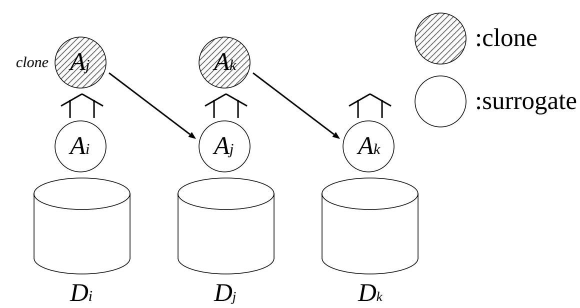

Figure 7. Creation of surrogate.

5.3. Commitment

In order to commit an agent, all or some of the surrogates are required to commit depending on the commitment condition. Each agent is also realized by using XA interface [11] which supports the two-phase commit protocol [Figure 5.3]. Each surrogate issues $prepare$ to a server on receipt of prepare from the agent. If $prepare$ is successfully performed, the surrogate sends a $prepared$ message to the agent. Here, the surrogate is referred to as $committable$. Otherwise, the surrogate aborts after sending $aborted$ to the agent. The agent receives responses from the agents after sending $prepare$ to the surrogates. On receipt of the responses, the agent makes a decision on *commit* or *abort* based on the termination condition. In the atomic condition, the agent sends $commit$ only if $prepared$ is received from every surrogate. The agent sends $abort$ to all commitment servers if $aborted$ is received from at least one surrogate. On receipt of $abort$, a committable surrogate aborts. In the at-least-one condition, the agent sends $commit$ to all committable servers only if $prepared$ is received from at least one server.

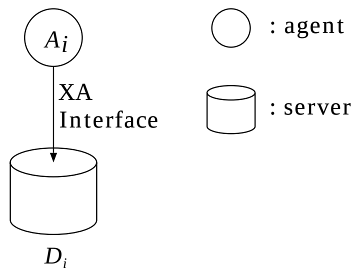

Figure 8. XA interface.

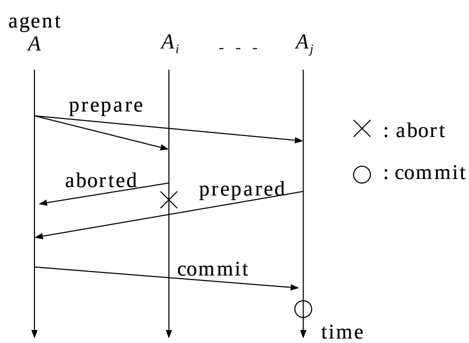

Figure 9. Conditional commitment.

Next, we discuss how to support robustness against agent failures. First, suppose a surrogate A_i of an agent A is faulty and recovered. Suppose a surrogate A_i is faulty after sending $prepared$. On recovery of the committable surrogate, the surrogate unilaterally commits if the surrogate is committable in the at-least-one transaction condition. In the atomic condition, the surrogate Ai asks the other surrogate if they had committed.

6. Concluding Remarks

This paper discussed a mobile agent model for processing 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. If the agent conflicts with other agents in an object server, the agent negotiates with the other agents. The negotiation is done based on the commitment conditions, i.e. all-or-nothing, at-least-one, majority, and $\binom{n}{r}$ conditions, and types of agents, i.e. ordered and unordered ones. We are now evaluating our mobile agent-based transaction systems for various types of applications.

References

[1] Aglets software development kit home. http://www.trl.ibm.com/aglets/.
[2] Jdbc data access api. http://java.sun.com/products/jdbc/.
[3] The source for java (tm) technology. http://java.sun.com/.
[4] Sybase sql server. http://www.sybase.com/.
[5] Oracle8i concepts vol. 1. Oracle Corporation, 1999. Release 8.1.5.
[6] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. In *Addison Wesley*, 1987.
[7] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of ACM*, 32(4):841–860, 1985.
[8] J. Gray and A. Reuter. Transaction processing : Concepts and techniques, 1993.
[9] A. N. S. Institute. Database language sql, 1986.
[10] F. H. Korth. Locking primitives in a database system. *Journal of ACM*, 30(1):55–79, 1989.
[11] X. C. Ltd. X/open cae specification distributed transaction processing: The xa specification., 1991. Document number XO/CAE/91/300.
[12] F. Mattern. Virtual time and global states of distributed systems, 1989. North-Holland, Amsterdam.
[13] A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf. Coordination of internet agents, 2001.
[14] I. Shimojo, T. Tachikawa, and M. Takizawa. M-ary commitment protocol with partially ordered domain. *Proc. of the 8th Int'l Conf. on Database and Expert Systems Applications(DEXA'97)*, pages 397–408, 1997.
[15] D. Skeen. Nonblocking commitment protocols. *Proc. of ACM SIGMOD*, pages 133–147, 1982.
[16] J. E. White. Telescript technology : The foundation for the electronic marketplace, 1994.