マルチメディア通信と分散処理　106－13
コンピュータセキュリティ　16－13
（２００２．２．１４）

Mobile Agent Model for Manipulating Distributed Objects Systems

Takao Komiya, Hiroyuki Ohsida, Katsuya Tanaka, and Makoto Takizawa
Tokyo Denki University, Japan
{komi, ohsida, katsu, taki}@takilab.k.dendai.ac.jp

This paper discusses mobile agents which manipulate objects in multiple object servers under various constraints among servers like atomicity and consistency. In database applications, application programs are performed on clients and issue requests to object servers. Then, the object servers send responses to the clients. On the other hand, programs named agents move to object servers where the agents manipulate objects in a mobile agent approach. If agents complete manipulating objects in the servers, the agents move to other servers. If an agent conflicts with other agents on an object server, the agents negotiate with each other to resolve the confliction. In this paper, we discuss how to perform application programs with different constraints on multiple object servers in the agent-based model.

分散オブジェクトシステムを操作するためのモバイルエージェントモデル

小宮 貴雄　大信田 裕之　田中 勝也　滝沢 誠
東京電機大学理工学部情報システム工学科

本論文はモバイルエージェントについて論じている。モバイルエージェントは原子性、一貫性のようなサーバ間の様々な制約下で、複数のオブジェクトサーバ上のオブジェクトを操作する。データベースアプリケーションで、アプリケーションプログラムは、オブジェクトサーバに要求を発行し、クライアント上で実行される。そのとき、そのオブジェクトサーバはそのクライアントに応答を送信する。一方、エージェントと名付けられたプログラムは、オブジェクトサーバを移動する。また、移動先のオブジェクトサーバ上でエージェントはモバイルエージェントアプローチでオブジェクトを操作する。もしエージェントがそのサーバ上でオブジェクトの操作を終えたならば、そのエージェントは他のサーバへ移動する。もし、あるエージェントがあるオブジェクトサーバ上で他のエージェントと競合するならば、そのエージェントは競合を解決するために他のエージェントと交渉する。この論文で、我々はエージェントベースなモデルで複数のオブジェクトサーバ上の異なった制約下で、どのようにアプリケーションプログラムを実行するのかについて論じる。

1 Introduction

In client-server database applications, application programs are performed on clients, which issue SQL [2] requests to object servers. The object servers send responses to the clients on completion of the requests. Requests and responses are exchanged among clients and servers in networks. The more number of requests are issued to object servers by applications and the more number of responses are sent back to the applications, the more communication overheads are increased. In the three-tier client-server architecture [4], applications move to application servers from clients in order to decrease the communication overheads between clients and servers.

In database applications, transactions are required to manipulate objects in object servers so as to satisfy ACID (atomicity, consistency, isolation, and durability) properties. [4]. For example, objects in multiple object servers are required to be atomically manipulated and transactions are serializable. In the traditional systems, objects are locked to realize the serializability [4] of transactions. In the locking protocol, multiple accesses to an object are coordinated based on a principle that only one transaction is a winner which can hold the object and the others are losers. There is another way like timestamp ordering [4]. Here, transactions are totally ordered in their timestamps. Transactions manipulate objects according to the timestamp order, i.e. the elder, the earlier. The locking protocol implies deadlock but no deadlock occurs in the timestamp ordering protocol.

In another computation paradigm, programs named mobile *agents* [1] move around data servers. First, an agent lands at a server and then is performed to manipulate data objects in the server. If the agent finishes manipulating the data objects in the server, the agent moves to another server which has data to be manipulated. Here, agents manipulate objects in object servers without exchanging messages in a network. Compared with traditional process-based applications like client-server applications, mobile agents have following characteristics;

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

In this paper, we discuss how to manipulate multiple object servers by using agents. Agents move around object servers without exchanging messages in the network. On the other hand, application programs and object servers are exchanging messages in the network. In addition, an agent negotiates with other agents if the 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 gives up to hold the objects.

In section 2, we present object servers. In section 3, we present an agent model for processing transactions. In section 4, we discuss how agents negotiate with other agents. In section 5, we discuss consensus conditions on which agents make an agreement in negotiation.

2 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.

Applications in clients initiate transactions in application servers. A transaction manipulates objects in one or more than one object server. A *transaction* T is an atomic sequence of methods for manipulating objects in object servers. A subsequence T_i of methods in T to manipulate objects in one object server D_i is referred to as *subtransaction* of T. A subtransaction T_i is also atomic sequence of methods in one object server D_i.

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

1. *begin-trans*: A subtransaction starts. A log for the subtransaction is initialized. Methods issued by the subtransaction are kept in record 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.

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. 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]. In order to realize the serializability, the locking protocol and timestamp ordering protocol [4] are used.

If a transaction manipulates objects in multiple object servers, the two-phase commitment protocol [4] is used to realize the atomic manipulation on multiple servers. After manipulating objects in the object servers by using methods, the transaction issues *prepare* messages to the servers. On receipt of *prepare*, update data of objects manipulated by the transaction is saved in the stable log of each server and then *yes* is sent back to the transaction. Unless succeeded in storing the update data in the log, the server sends *no* to the transaction and the subtransaction on the server aborts. Then, the transaction issues *commit* to the servers only if the transaction receives *yes* from all the servers. Otherwise, the transaction issues *abort* to every server which has sent *yes*. On receipt of *abort*, the log is removed and the subtransaction aborts.

Object servers may be replicated in order to make the system more reliable and available. Suppose servers D_{j1}, ..., D_{jm} ($m \geq 2$) are replicas of an object server D_j. A collection of the replicas $\{D_{j1}$, ..., $D_{jm}\}$ is referred to as *cluster* of D_j, denoted as $C(D_j)$.

3 Agents

3.1 Computation model

An agent is a procedure which can be 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. Every object server is assumed to support a platform to perform agents.

First, an agent A is initiated by an application or 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 in order to perform the agent A on D_i. If enough resource like memory to perform the agent A is allocated for the agent A on the server D_i, the agent A can be performed. Here, D_i is referred to as *current* server of A and the agent A is referred to as *land at* the server D_j. Objects in the server D_i are manipulated by the agent A through methods. In result, state of object may be changed and a part of the state may be derived. Data derived from the server D_i may be stored in the agent A. Thus, an instance A_i of the agent A on the object server D_i shows a subtransaction, i.e. a sequence of methods for manipulating objects in the server D_i. Then, the agent A finds another server D_j which has objects to be manipulated by A. Then, the agent A moves to the server D_j. Here, the agent A may carry objects obtained from D_i as the data of A [Figure 1]. If enough resource like memory in the server D_j is allocated for the agent A, A lands at D_j.

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. For example, A_1 issues a method *reset* and A_2 issues *increment* to a *counter* object in a server D_j. Here, A_1 and A_2 conflict. The agent A is allowed to land at D_j if the following condition is satisfied:

[Landing conditions]

1. Enough resource to perform an agent A is allocated for the agent A in an object server D_j.
2. There is no agent on D_j which conflicts with A.

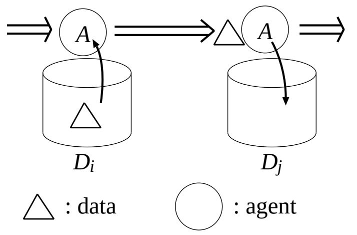

Figure 1: Agent.

3.2 Movement of agent

Suppose an agent A is at an object server D_i and is finding an object server where the agent A can land. 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 servers $\{D_{j1}, \ldots, D_{jm}\}$ at which an agent A can land from a server D_i. For example, there are replicas $D_{j1}, \ldots, D_{jm}$ of some server D_j. $Cand_i(A)$ is a cluster $C(D_j)$ of the replicas. If an agent A only reads objects, one server, i.e. one replica D_{jk} is selected and then moves to the server D_{jk}. Here, an agent A takes another replica D_{jk} in the candidate set $Cand_i(A)$. If the agent A updates objects, all the servers in $C(D_j)$ are taken and replicas in all the servers are manipulated by A. This is similar to a famous two-phase locking (2PL) protocol [4]. On the other hand, an agent A issuing *read* takes a subset Q_r of the candidate set $Cand_i(A)$, which is a *read* quorum. The agent A issues *write* to servers in a *write* quorum Q_w. Here, $Q_r \cap Q_w \neq \phi$ and $Q_r \cup Q_w = Cand_i(A)$. This shows a quorum-based protocol [5].

In another case, the agent A is composed of multiple modules $A_1, \ldots, A_m$ ($m > 1$) which can be performed in any order and concurrently. Here, each module A_h can be performed on a server D_{jh} ($h = 1, \ldots, m$). As presented in the examples, there are two cases with respect to how many servers to be taken by an agent A at a server D_i:

1. One server in the candidate set $Cand_i(A)$ is taken.
2. Multiple servers in $Cand_i(A)$ are taken.

In the first case, we have to discuss which server in the candidate set $Cand_i(A)$ to be taken. For example, a server D_{jk} which is nearest to D_i is taken. A server which is least loaded can be also taken.

In the second case, multiple servers, possibly all the servers in the candidate set $Cand_i(A)$ are taken. In addition to discussing which servers to be taken in $Cand_i(A)$, we have to discuss how to find an optimal route to visit all the servers in the candidate set $Cand_i(A)$ [Figure 2]. For example, a route whose communication cost is the minimum is selected. This shows a serial computation. In another way, the agent A can be splitted into multiple *subagents* $A_1, \ldots, A_m$ [Figure 3]. Each subagent A_k is issued to a server D_{jk} in $Cand_i(A)$. The subagents $A_1, \ldots, A_m$ are concurrently performed on object servers. After manipulating objects in the servers, the subagents are merged into one agent A again. Each subagent A_k might bring data d_k obtained from an object server D_{jk}. We have to discuss where all the subagents are merged into an agent A. One idea is to take one object server D_{jk} where a subagent A_k is performed and the data d_k is the largest in all the subagents $A_1, \ldots, A_m$.

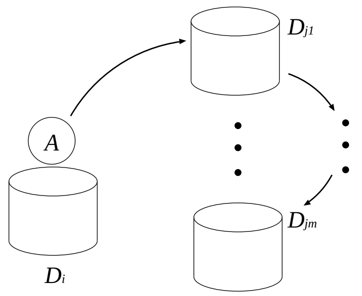

Figure 2: Optimal routing.

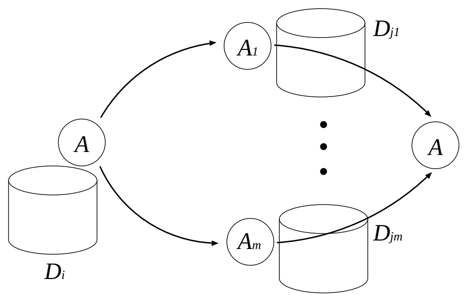

Figure 3: Split and merge of agents.

3.3 Operations on agents

As discussed here, agents are moving, splitted to multiple agents, and merged into one agent. Following operations on agents are supported by each object server:

1. A = *create-agent*( ): a new agent is created. An object server where an agent A is created is referred to as *home* server of A.
2. A' = *clone-agent*(A): a clone A' of an agent A is created.
3. *split-agent*(A, $\{A_1, \ldots, A_m\}$): one agent A is splitted into multiple subagents $A_1, \ldots, A_m$ ($m \geq 1$).
4. *merge-agent*($\{A_1, \ldots, A_m\}$, A): multiple agents $A_1, \ldots, A_m$ are merged into an agent A.
5. *annihilate-agent*(A): an agent A is destroyed.
6. C = *cand-agent*(A, D_i): a candidate set $C = Cand_i(A)$ is obtained.
7. D_j = *select-agent*(A, D_i, C, S): one server D_j is selected from a candidate set C according to the strategy S. If $S = One$, an optimal server D_j is selected in C. If $S = All$, a server D_j to be visited from D_j is selected according to an optimal writing strategy. All the servers in the candidate set C are to be visited.
8. *move-agent*(A, D_i, D_j): an agent A in a server D_i is moved to another a server D_j.
9. *negotiate-agent*(A, $\{A_1, \ldots, A_m\}$): an agent A negotiates with other agents $A_1, \ldots, A_m$ to make some agreement. A decision *do*, *abort*, *block*, or *retreat* is returned.
10. *land-agent*(A, D_i): if an agent A can land at a server D_i, *true* is returned. Otherwise, *false*.
11. *conflict-agent*(A, D_i): if there is another agent which conflicts with an agent A in a server D_i, *true* is returned. Otherwise, *false*.
12. *term-agent*(A): if an agent A can finish, *yes* is returned. Otherwise, *no* is returned.
13. *commit-agent*(A): if an agent A satisfies commitment condition *A commits*. Otherwise, *A aborts*.

An agent A is performed on an object server D as follows;

1. An agent A is created at a home server D, i.e. A = *create-agent*(D).
2. After the agent A is performed in a current server D, A moves to another server if the termination condition *term-agent*(A) is not satisfied. A candidate set C is obtained; C = *cand-agent*(A, D).
3. If *parallel* strategy is taken, an agent A is splitted to agents $A_1, \ldots, A_n$, i.e. *split-agent*(A, $\{A_1, \ldots, A_n\}$). Then, $A_1, \ldots, A_n$ move to candidate servers in C.
4. If *serial* strategy is taken, a destination server D_i of the agent A is decided, i.e. D_i = *route-agent*(A, D).
5. If the agent A could land at the destination server D_i, i.e. *land-agent*(A, D_i) is true, the agent A moves to a server D_i.
6. If the agent A does not conflict with agents $A_1, \ldots, A_n$ in D_i, i.e. *conflict-agent*(A, D_i) is *false*, A is performed on D_i. Otherwise, A negotiates with the agents $A_1, \ldots, A_n$ conflicting with A, *negotiate-agent*(A, $\{A_1, \ldots, A_n\}$). If *do* is returned, A starts. The agent A is started, aborted, and blocks on the server D_i if *do*, *abort*, and *block* are returned, respectively.
7. If the agent A is successfully performed on the server D_i, a *surrogate* agent A_i of A is created and resides at D_i, A_i *clone-agent*(A). If there is no other destination, i.e. all the object servers are manipulated, *term-agent*(A) is true, the commitment procedure is started based on the consensus condition *Cons*(A). Otherwise, go to 2.

4 Consensus among Agents

An agent A manipulates objects in multiple object servers $D_1, \ldots, D_m$ ($m > 1$). After finishing manipulating objects in all the object servers, the agent A commits if some condition C on the servers $D_1, \ldots, D_m$ is satisfied. Otherwise, A aborts. For example, an atomic all-or-nothing condition is used to realize the atomicity of a transaction. That is, the agent commits only if all the object servers are successfully updated. Otherwise, the agent aborts, i.e. no update is done on the objects in any object server. The two-phase commitment (2PC) protocol [4] is used to realize the all-or-nothing principle in distributed database systems. In another example, an application would like to book one hotel. The application issues a booking request to multiple hotel objects. Here, the application can commit if at least one hotel object is obtained. Thus, if at least one of the servers is successfully manipulated, the agent A commits. There are following consensus conditions;

[Consensus conditions]

1. *Atomic consensus*: an agent is successfully performed on all the object servers, i.e. all-or-nothing principle. This is a condition used in the traditional two-phase commitment protocol.
2. *Majority consensus*: an agent is successfully performed on more than half of 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 out of n servers ($r \leq n$).

The atomic, majority, and at-least-one consensus conditions are shown in forms of $\binom{n}{n}$, $\binom{n}{\ulcorner (n+1)/2 \urcorner}$, and $\binom{n}{1}$ consensus ones, respectively. More general consensus conditions are discussed in a paper [8]. Each agent A is assumed to have a consensus condition *Cons*(A) given by an application.

Suppose an agent A finishes manipulating objects in object servers $D_1, \ldots, D_m$. Let A_i be a *surrogate* agent of the agent A in an object server D_i [Figure 4]. Suppose another agent B might come to D_j after the agent A leaves an object server D_j. Here, the agent B negotiates with the surrogate agent A_i of the agent A if B conflicts with 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. The agent A starts the negotiation procedure with its surrogates $A_1, \ldots, A_m$. If a consensus condition C on $A_1, \ldots, A_m$ is satisfied, the agent A commits. For example, an agent commits if all the surrogates safely exist in the atomic consensus condition. As discussed in a following section, surrogates do negotiation with agents. Then, the surrogate abort. If the surrogates exit, the computation performed by the agent is successful. Then, the surrogate agents of A are annihilated. Here, other agents conflicting with the agent A are allowed to manipulate objects.

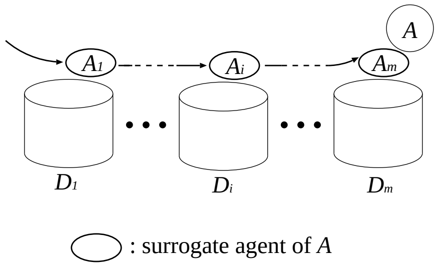

Figure 4: Surrogate agents.

5 Negotiation Strategies

5.1 Protocol

Unless the landing conditions are satisfied, the agent A can not land at the server D_j. Here, the agent A can take one of the following ways:

1. The agent A waits in the current object server D_i.
2. The agent A finds another object server D_k which has objects to be possibly manipulated before D_j by A.
3. The agent A negotiates with other agents in D_j which hold resources.
4. The agent A *aborts*.

Suppose an agent lands at a current object server D_i. Here, there might be other agents $B_1, \ldots, B_k$ which are being performed on the object server D_i. Each agent B_i is an agent or surrogate agent of an agent. If the agent A conflicts with some agent B_j on an object o, A negotiates with B_j with respect to which agent A or B_j holds the object o. There are following negotiation strategies:

1. The agent A blocks until the agent B_j commits.
2. The agent A takes over B_j, i.e. B_j releases the objects and blocks. Then A starts.
3. B_j aborts and 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 B waits for release of an object held by A, A and B are deadlocked. Thus, deadlock among agents may occur. When an agent A blocks in a 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.
2. Every surrogate A_j of 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 A conflicts with B_j and B_j holds an object. Here, A starts to do the negotiation with an agent B_j on D_j by using a following negotiation protocol :

[Negotiation protocol]

1. An agent A sends a *can-I-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 with a method op in an object server D_j.
2. On receipt of $CIU(o, op)$ from an agent A, an agent B_j sends OK to A if B_j can release the object o or B_j does not mind if A manipulates the object o. Here, there are two approaches to B_j's releasing the object o :
   a. B_j aborts if A precedes B_j.
   b. B_j rolls back to a checkpoint and then restarts if A precedes B_j. Otherwise, B_j sends No to A.
3. On receipt of OK from B_j, A starts manipulating the object o.
4. On receipt of No from B_j, there are following ways:
   a. A blocks until A receives OK/NO from B_j.
   b. A aborts.
   c. A triggers the second level negotiation protocol. □

If the agent B_j agrees with the agent A in the negotiation protocol, A can manipulate objects by taking over B_j. In the second way, the agent B_j not only releases the object but also aborts.

5.2 Resolution of confliction

There are two types of agents;

1. Ordered agents.
2. Unordered agents.

Every pair of ordered agents manipulate objects in a well-defined way. Agents are ordered. Each 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* [4] can be used as an identifier of an agent. That is, the identifier $pid(A)$ of an agent A is time $ts(A)$ when 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. If a logical clock like vector clock [6] is used as precedent identifier, the agents are partially ordered. An agent A_1 is concurrent with another agent A_2 iff neither A_1 precedes A_2 nor A_2 precedes A_1.

Suppose multiple agents $A_1, \ldots, A_m (m>1)$ would like to manipulate an object o in an object server D_i and 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_s and A_t are concurrent, A_s and A_t are allowed to be performed in any order. However, if A_s and A_t conflict on a pair of servers D_i and D_j. A_s and A_t are required to be performed in a same order at D_i and D_j. There never occurs deadlock.

Unordered agents are not ordered. Like locking protocols, an unordered agent had obtained 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 D_2 and is moving to D_1. If A_1 and A_2 conflict on each of D_1 and D_2, neither A_1 can land at D_2 nor A_2 can land at D_1. Here, deadlock occurs.

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

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

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

6 Concluding Remarks

This paper discussed an agent model for 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, the agent negotiates with the other agents. The negotiation is done based on the commitment conditions and types of agents, i.e. ordered and an ordered.

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] Arai, K., Tanaka, K., and Takizawa, M., "Group Protocol for Quorum-Based Replication," *Proc. of IEEE ICPADS'00*, 2000, pp.57-64.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] Skeen, D., "Nonblocking Commitment Protocols," ACM SIGMOD, 1982, pp.133-147.

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