Mobile Agent Model for Distributed Objects System

Tomoya ENOKIDO and Makoto TAKIZAWA
Department of Computers and Systems Engineering
Graduate School of Science and Engineering, Tokyo Denki University, Japan

ABSTRACT: A transactional agent is a mobile agent which manipulates objects in one or more than one computer so as to satisfy some constraint like ACID. An agent creates a surrogate agent on a computer on leaving the computer. A surrogate holds objects manipulated by the agent until the agent terminates. The surrogate can recreate a new incarnation of the agent if the agent is faulty. Transactional agents for multiple database servers are implemented in java Aglets. We evaluate the performance of transactional agent model in terms of total access time compared with traditional client-server model.

1. Introduction

In traditional client-server applications, application programs and objects are distributed into clients and servers, respectively. On the other hand, any computer named servant is peer which can have programs and objects in peer-to-peer (P2P) systems. Huge number and various types of computers are interconnected in networks, ranging from high-performance, high-reliable computers to less-reliable personal computers and mobile stations. It is critical to discuss how to develop applications on databases in P2P systems. In database applications, transactions manipulate objects so as to satisfy ACID (atomicity, consistency, isolation, and durability) properties.

Mobile agents [1, 2, 3, 4, 5] locally manipulate objects on computers by moving to the computers. An agent first moves to a computer where objects are manipulated. After manipulating all or some computers, an agent makes a decision on commit or abort. A transaction with ACID property initiates a subtransaction on each database server, which is realized in mobile agents [1, 2, 3, 4, 5]. In this paper, a transactional agent is defined to be a mobile agent which moves around computers and locally manipulates objects in one or more than one computer with type of commitment constraint. In addition, an agent negotiates with another agent if the agents manipulate a same object in a conflicting manner. Through the negotiation, each agent autonomously makes a decision on whether the agent holds or releases the objects. A transactional agent autonomously finds another destination computer if a computer where the agent to move is faulty.

If an agent leaves a computer, the objects locked by the agent are automatically released by the operating system. Hence, an agent creates a surrogate agent which holds locks on objects in a computer on behalf of the agent after the agent leaves the computer. The surrogate agent holds locks on objects until the agent commits or aborts. The agent is faulty due to the fault of a current computer where the agent exists. Some surrogate of the agent which exists on another computer recreates a new incarnation of the agent. We discuss how to design and implement transactional agents manipulating objects in multiple computers in presence of computer faults by using Java aglets [6].

In section 2, we present a system model. In section 3, we discuss transactional agents. In section 4, we discuss implementation of transactional agents. In section 5, transactional agents are evaluated compared with client-server applications.

2. System Model

A system is composed of computers interconnected in reliable networks. Each computer is equipped with a class base (CB) where classes are stored and an object base (OB) where a collection of persistent objects are stored. A class is composed of attributes and methods. An object is an instantiation of a class. That is, an object is an encapsulation of data and methods for manipulating the data. Suppose an object \( o \) supports a pair of methods \( op_1 \) and \( op_2 \). If result obtained by performing the methods \( op_1 \) and \( op_2 \) on an object 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 \( o \). 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 \( op_u \) and \( op_v \) conflict, every method \( op_j \) from \( T_j \) has to be performed before every method \( op_u \) from \( T_2 \) conflicting with the method \( op_u \). This is a \emph{serializability} property. There are locking protocols and timestamp ordering protocols to realize the serializability. Each computer supports an agent with an \emph{isolation} level which shows when the agent releases objects. In the locking protocol, a transaction locks an object before manipulating the object.

A \emph{mobile agent} is an autonomous program which is composed of a \emph{kernel class} and \emph{shell classes}. The kernel class of an agent moves around computers in networks. Methods in shell classes are invoked by methods in the kernel class. Shell classes mean not only application-specific classes but also library classes like JDBC classes and JAVA classes. JDBC and JAVA classes are used to manipulate objects, respectively.

A \emph{home computer} \( \text{home}(c) \) of a class \( c \) is a computer where the class \( c \) is stored in the class base \((CB)\). A home computer \( \text{home}(A) \) of a mobile agent \( A \) is a home computer of the kernel class of the \( A \). If a method in a class \( c \) is invoked, the class \( c \) has to be found in the system. Each class has a unique identifier in the system.

![Fig.1. Computer.](image1)

Fig.1. Computer.

Each computer is assumed to support a platform to perform a mobile agent on an object base \((OB)\) [Fig.1]. A platform includes \emph{cache} and \emph{class base} \((CB)\). A class base \((CB)\) is composed of classes. The kernel and shell classes of an agent \( A \) are stored in the class base \((CB)\) of the home computer \( \text{home}(A) \). If the agent \( A \) invokes a method \( t \) of a class \( c \) in a computer, the class \( c \) is loaded from the home computer \( \text{home}(c) \) to the cache in the computer [Fig.2]. Then, the method \( t \) of the class \( c \) is performed in the computer. If a method \( u \) of another class \( d \) is invoked in the method \( t \), the class \( d \) is loaded from the home computer \( \text{home}(d) \) as well as the class \( c \). Meanwhile, if another agent \( B \) invokes a method \( op \) of the class \( c \) in the computer, the class \( c \) in the cache is used to invoke the method \( op \) without loading the class \( c \). Thus, if classes are cached in a computer, methods in the classes are locally invoked in the computer without any communication. Otherwise, it takes a longer time to invoke methods since classes with the methods are transferred from the home computers in networks.

An agent \( A \) is initiated in a computer autonomously or by an application. First, a kernel class of the agent \( A \) is loaded to the computer. Then the agent is started on the computer. The computer is referred to as \emph{base computer} of the agent \( A \), denoted \( \text{base}(A) \). The base computer \( \text{base}(A) \) is not necessarily the same as the home computer \( \text{home}(A) \). The agent \( A \) leaves the base computer for a computer \( D \) to manipulate objects. The agent \( A \) is performed on the computer \( D \). Here, \( D \) is a current computer of the agent \( A \), denoted \emph{current}(\( A \)).

If the agent \( A \) invokes a method \( t \) of a class \( c \), the cache is first searched for the class \( c \). If the class \( c \) is found in the cache, the method \( t \) of the class \( c \) in the cache is invoked. If not, class base \((CB)\) of the computer \( D \) is locally searched. If the class \( c \) is found in the class base \( CB \), the class \( c \) in \( CB \) is taken to invoke the method \( t \). Otherwise, the class \( c \) is transferred from the home computer \( \text{home}(c) \) into the computer \( D \). Here, the class \( c \) is loaded to the agent \( A \), i.e., \emph{cached} in the computer \( D \). The method \( t \) of the class \( c \) is performed on \( D \). If another agent \( B \) comes to \( D \) after the agent \( A \) has left \( D \), the agent \( B \) can take usage of the class \( c \) in the cache by neither communicating with the home computer \( \text{home}(c) \) nor having access to the local class base \( CB \).

Thus, if classes which an agent \( A \) uses are already loaded in the cache, the agent is efficiently performed on a computer. Otherwise, it takes a longer time to transfer classes from the home computers to the current computer \emph{current}(\( A \)). The agent \( A \) manipulates objects in the object base \( OB \) of the current computer \( D \) by using the classes loaded.

![Fig.2. Home computer.](image2)

Fig.2. Home computer.

![Fig.3. Ways to load classes.](image3)

Fig.3. Ways to load classes.
There are two ways to load classes of an agent \( A \) from a home computer [Fig.3]. One way is an interactive where one a class \( c \) is loaded from a home computer \( \text{home}(A) \) each time a method of the class \( c \) is invoked by the agent \( A \). Another way is a batch one where a collection of multiple classes are loaded.

3. Transactional Agents

3.1 Commitment conditions

A transaction in client-server model sends requests to and receives responses from servers. On the other hand, a transactional agent [1, 2, 3, 4, 5] locally manipulates objects in computers by moving around in networks. A scope \( \text{Scp}(A) \) of an agent \( A \) means a set of computers where the agent \( A \) possibly manipulates objects. If the agent \( A \) finishes manipulating objects in each computer, the commitment condition \( \text{Com}(A) \) of \( A \) is checked. There are following commitment conditions:

1. \( \text{Atomic commitment} \) an agent is successfully performed on all the computers.
2. \( \text{Majority commitment} \) an agent is successfully performed on more than half of the computers.
3. \( \text{At-least-one commitment} \) an agent is successfully performed on at least one computer.
4. \( (\frac{r}{n}) \text{ commitment} \) an agent is successfully performed on more than \( r \) out of \( n \) object servers \( (r \leq n) \).

A commitment condition \( \text{Com}(A) \) is specified for each agent \( A \) by an application. There are still discussions on when the condition \( \text{Com}(A) \) of an agent \( A \) can be checked while the agent \( A \) is moving around computers. Let \( \text{Hist}(A) \) be a history of \( A \), i.e. set of computers where \( A \) has so far manipulated objects \( (\text{Hist}(A) \subseteq \text{Scp}(A)) \).

Fig.4, Surrogate agents.

Suppose an agent \( A \) leaves a computer \( D \). An agent named \( \text{surrogate} \) is created and then is left on \( D \) [Fig.4]. The surrogate agent \( A \) still holds objects in \( D \) which are manipulated by \( A \) on behalf of \( A \). A surrogate agent releases objects depending on the isolation level of \( A \). The surrogate agent releases objects until \( A \) commits or aborts at the highest isolation level. The surrogate agent releases objects just after leaving \( D \) at the weakest isolation level. Surrogates commit or abort according to the decision of the agent. Surrogates created by \( A \) are referred to as \( \text{sibling surrogates} \).

Suppose another agent \( B \) arrives at a computer \( D \) after \( A \) leaves \( D \). Here, the agent \( B \) negotiates with the surrogate agent \( A \) of \( A \) in \( D \) if \( B \) conflicts with \( A \). After the negotiation, the agent \( B \) might take over the surrogate \( A \). Thus, when the agent \( A \) finishes visiting all the computers, some surrogate may not exist due to the fault and abortion in negotiation with other agents.

A starts the negotiation procedure with its surrogates \( A_i, ..., A_m \). If a commitment condition \( \text{Com}(A) \) on the surrogates \( A_1, ..., A_m \) is satisfied, \( A \) commits.

3.2 Resolution of confliction

Suppose an agent \( A \) moves to a computer \( D \) from another computer \( D \). The agent \( A \) cannot be performed on \( D \) if there is an agent or surrogate \( B \) conflicting with \( A \). Here, the agent \( A \) can take one of the following ways:

1. \( \text{Wait} \) The agent \( A \) in the computer \( D \) waits until the agent \( A \) can land at a computer \( D \).
2. \( \text{Escape} \) The agent \( A \) finds another computer \( D \) which has objects to be possibly manipulated before the computer \( D \).
3. \( \text{Negotiate} \) The agent \( A \) negotiates with the agent \( B \) in the computer \( D \). After the negotiation, the agent \( A \) takes over \( B \), i.e. \( B \) releases the object or abort.
4. \( \text{Abort} \) The agent \( A \) aborts.

Fig.5, Retreatment.

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. If the timer expires, the agent \( A \) takes a following way:

1. The agent \( A \) retreats to a computer \( D \) in the history \( \text{Hist}(A) \). All the surrogates of \( A \) which have been performed after performed on \( D \) are aborted.
2. Then, the surrogate agent \( A \) on \( D \) recreates a new incarnation of the agent \( A \). The agent \( A \) finds another destination computer \( D \) [Fig.5].

The surrogate \( A \), to which the agent \( A \) retreats plays a role of checkpoint of \( A \). Differently from traditional checkpoints, the agent \( A \) retracting to some surrogate \( A \)
autonomously finds an operational computer which may be different from one which the agent \( A \) has visited.

Suppose a surrogate agent \( B \) holds an object in a computer \( D_i \). An agent \( A \) would like to manipulate the object but conflicts with the surrogate agent \( B \) in \( D_i \). The surrogate agent \( B \) makes a following decision depending on the commitment conditions of \( B \):

1. **Atomic commitment** The agent \( A \) waits until the surrogate \( B \) finishes.
2. **At-least-one commitment** If the surrogate \( B \) knows at least one sibling surrogate of \( B \) is committable, the surrogate \( B \) releases the object and aborts. The surrogate \( B \) informs the other sibling surrogates of this abort.
3. **Majority commitment** If the surrogate \( B \) knows more than half of the sibling surrogates are committable, the surrogate \( B \) releases the object and aborts. The surrogate \( B \) informs the other surrogates of this abort.
4. \( \left( \frac{n}{r} \right) \) commitment If the surrogate \( B \) knows more than or equal to \( r \) sibling surrogate agents are committable, the surrogate \( B \) releases the object and aborts.

4. **Implementation of Transactional Agent**

We discuss how to realize transactional agents on multiple relational database servers. A transactional agent is composed of a **kernel class** and **shell classes**. An object base \( (OB) \) is realized in a relational database system, Oracle and Sybase. An agent manipulates table objects by issuing SQL commands, i.e. **select**, **insert**, **delete**, and **update** in a current computer \( D_i \). The computation of each agent \( A \) on a computer \( D_i \) is realized as a **transaction** on a database system in \( D_i \). If an agent \( A \) leaves \( D_i \), the transaction for the agent \( A \) commits or aborts. That is, objects manipulated by the agent \( A \) are released and then can be manipulated by other agents. Even if the agent \( A \) leaves \( D_i \), objects manipulated by the agent \( A \) are required to be held by \( A \) because the agent \( A \) may abort after leaving \( D_i \). If the objects are released, the transactional agent may be **unrecoverable**. Therefore, a surrogate agent is newly introduced as discussed in the previous section. The surrogate is realized by a **clone** of the agent and an **object agent** \( OBA_i \). Each object agent \( (OBA) \) behaves as follows:

1. On arrival at a computer \( D_i \), an agent \( A \) initiates an object agent \( OBA_i \) on \( D_i \). \( OBA_i \) initiates a transaction on an object base \( OB \), i.e. database system in \( D_i \) by issuing a **start-transaction** command.
2. If the agent \( A \) issues a method for manipulating objects, the object agent \( OBA_i \) issues SQL commands to the database system in \( D_i \) on behalf of \( A \).
3. If the agent \( A \) finishes, the agent \( A \) leaves the computer \( D_i \). However, \( OBA_i \) is still operational and holding the objects in \( D_i \).
4. The object agent \( OBA_i \) commits and aborts if the agent \( A \) sends **commit** and **abort** requests to \( D_i \), respectively. Thus, \( OBA_i \) communicates with \( A \).

An object agent \( OBA_i \) stays on a computer \( D_i \) while holding objects even if an agent \( A \) leaves \( D_i \). \( OBA_i \) is considered to be a local transaction on an object base \( OB \). On completion of \( A \), \( OBA_i \) is terminated.
follows:
1. A clone $A'$ of the agent $A$ is created in the computer $D_i$.
2. The clone $A'$ leaves the computer $D_i$ but the agent $A$ stays in $D_i$.

If an agent $A$ leaves a computer $D_i$, objects held by $A$ are released. Hence, the agent $A$ stays on the computer $D_i$ while the clone $A'$ leaves $D_i$. The agent $A$ which stays in $D_i$ and the object agent $OBA_i$ plays a role of surrogate agent of $A$. On the other hand, the clone agent $A'$ moves to another object takes over the surrogate [Fig.7].

An agent $A$ can commit if all or some of the surrogates commit depending on the commitment condition $Conf(A)$. Communication among an agent and its surrogate agents is realized by using the XA interface which supports the two-phase commitment protocol [Fig.6]. Each surrogate agent issues a prepare request to a computer on receipt of a prepare message from the agent. If prepare is successfully performed, the surrogate agent sends a prepared message to the agent. Here, the surrogate is committable. Otherwise, the surrogate agent aborts after sending aborted to the agent. The agent receives responses from the surrogate agents after sending prepare to the surrogates. On receipt of the responses from surrogates, the agent makes a decision on commit or abort based on the commitment condition. For example, if the atomic condition holds, the agent sends commit only if prepared is received from every surrogate. The agent sends abort to all committable agents if an aborted message is received from at least one surrogate. On receipt of abort, a committable surrogate aborts. In the at-least-one commitment condition, the agent sends commit to all committable surrogates only if prepare is received from at least one surrogate.

Next, we discuss how to support robustness against agent faults. Suppose a surrogate agent $A_i$ of an agent $A$ is faulty after sending prepared. Here, the surrogate agent is committable. On recovery of the committable surrogate, the surrogate unilaterally commits if the surrogate is committable in the at-least-one commitment condition. In the atomic condition, the surrogate $A_i$ asks the other surrogates if they had committed. Suppose the surrogate $A_i$ is abortable, i.e. faulty before receiving prepared. On recovery of $A_i$, $A_i$ unilaterally aborts.

5. Evaluation
We evaluate performance of a transactional agent in terms of total access time compared with client-server model. In the evaluation, there are three computers $D_1$, $D_2$, and $D_3$ which are a Sun workstation (SPARC 900MHz x 2) and a pair of Windows PCs (Pentium3 1.0GHz x 2 and Pentium3 500MHz), respectively. The computers are interconnected with 100base Ethernet. JDBC classes are initially loaded in each computer. Object bases in the computers are realized in Oracle.

An application program $A$ manipulates table objects by issuing select and update to some number of computers at the highest isolation level, i.e. select for update in Oracle. The program is implemented in Aglets for a transactional agent (TA) model and JAVA for a client-server (CS) model. In the client-server model, the application program $A$ is performed on a client computer and application server in 2-tier (2CS) and 3-tier (3CS) models, respectively. In the transactional agent model, the application program $A$ is realized in Aglets. The computation of transactional agent is composed of five steps: movement, loading class, object manipulation, clone creation, and commitment steps. In the client-server model, there are four computation steps of program initialization, class loading to client, object manipulation, and two-phase commitment. Fig.8 shows how long it takes to perform each step for two cases, one for manipulating one computer and another for manipulating two computers, in client-server (CS) and transactional agent (TA) models. In the transactional agent (TA) model, Aglets classes are loaded to each computer before an agent is performed. Since Java classes are loaded to clients in the client-server model, it takes a constant time to load the classes independently of number of computers. As shown in Fig.8, it takes a shorter time to manipulate objects in a transactional agent than the client-server model because there is no communication between agent and computer. The time to load Aglets classes in each computer is about half of the total computation time in the transactional agent. The transactional agent more shortly manipulates objects than the client-server model. On the other hand, Aglets classes have to be loaded in the transactional agent. It takes about two seconds to load Aglets classes.

The total access time shows a duration from time when the application program starts until time when the application program ends. Next, the total access time is measured for transactional agent (TA) and
client-server (CS) models. A pair of applications are taken for evaluation. In the first application, one record is derived from each object base. In the second one, 100 thousands records are derived from each object base, i.e. Fig.9 and 10 show the first and second cases, respectively for 2CS and 3CS models. NC means that classes are not cached for program. Here, the classes are loaded from home computers to a current computer before a program or agent is performed.

The more number of records are manipulated, the shorter response time the transactional agent supports than the client-server model. In the transactional agent (TA) model, an object agent (OBA) is created each time an agent is performed on a computer. It takes time, about 100 [msec] to create OBA in a computer. We are now implementing another way that OBA is always operational and every agent communicates with OBA in each computer. Each agent can manipulate objects without newly creating OBA.

6. Concluding Remarks
The authors discussed a transactional agent model to manipulate objects in multiple computers with types of commitment constraints. An transactional agent first moves to a computer and then locally manipulates objects. The agent autonomously moves around the computers. We discussed how to implement transactional agents in Aglets and Oracle. We showed the evaluation of the mobile agent-based transaction systems for applications. If Aglets classes are a priori loaded, the transactional agents can manipulate object servers more shortly than the client-server model.

Fig.8, Access time.

Fig.9, Client server(CS) vs. transactional agent(TA).

REFERENCES

Part of this research was supported by the Research Institute for Science and Technology, Tokyo Denki University, with the number Q02J-05.