Vol. 37 No. 8 Transactions of Information Processing Society of Japan Aug. 1996

Regular Paper

Transactions on Distributed Mobile Replicated Objects

TAKEAKI YOSHIDA[†] and MAKOTO TAKIZAWA[†]

According to the advances of communication technologies, various kinds of mobile wireless stations like personal handy systems and intelligent robots are available. Objects support abstract operations and are distributed in not only fixed stations but also mobile ones. Transactions manipulate multiple, possibly replicated objects in mobile and fixed stations. While the objects are moving from one location to others in the system, the quality of service (QoS) supported by the objects change. The connection is tentatively closed by the mobile station in order to reduce power consumption while the operations issued by the mobile station are being computed, i.e., disconnected operations. We discuss the migration and replication methods to treat disconnected operations. In addition, we present an optimistic concurrency control to maintain mutual consistency among the replicas by taking into account more abstract types of operations on the objects other than read and write on files.

1. Introduction

According to the advances of communication and computer technologies, kinds of mobile wireless stations like *personal handy systems* are available. The distributed systems are composed of mobile and fixed stations interconnected by communication networks. The fixed stations are connected at a fixed location in the communication network. The mobile stations in a *cell* communicate with the *mobile support station* (*MSS*) in the cell by using wireless communication. The mobile support station maintains the *connection* of the mobile station in the cell with another station. If the mobile station moves to another cell, it can continue to communicate with the station through the mobile support station in the cell. Tanaka[17) and Teraoka[18) discuss protocols for supporting connections with mobile stations.

Users access objects in the fixed server stations through the mobile stations. The mobile stations are not equipped with enough battery capacity to have long-time communication. In order to reduce the power consumption, the connections between the mobile stations and fixed stations are disconnected while the operations issued by the mobile stations are being computed, i.e., *disconnected* operations[12). One technique to compute the disconnected operations is to *cache* data in the fixed station like a server to the mobile station. Without communicating with the fixed station, users can manipulate the data cached into the mobile station. Barbara[3) and Huang[9) present how to cache the data in the fixed stations to the mobile stations and how to maintain the mutual consistency among the caches and the fixed stations. Jing[11) discusses the locking scheme based on the optimistic two-phase locking[4) on the replicas and a way to reduce the communication overhead to release the locks.

In this paper, the distributed system is assumed to be composed of objects distributed in multiple stations. Each object supports abstract data and operations for manipulating the data, while only *read* and *write* operations are considered in the other papers[3),9),11). On receipt of the operations, the objects start to compute the operations, which furthermore may issue operations to other objects. On completion of the operations, the objects send back the responses. The computation of each operation on an object is viewed to be *atomic*, i.e., the operation is completely computed or nothing[2),7). The computation of an operation issued by the operation is also atomic, i.e., *nested*[16),19).

The objects may be replicated into multiple replicas in order to increase the reliability, availability, and performance. In this paper, we assume that the object is fully replicated, i.e., the replicas have the same data and operations as the object. We discuss an optimistic concurrency control to maintain the mutual consistency among replicas of objects supporting more abstract nested operations than read and write.

According to the movement of the mobile stations, the objects in the mobile stations are

---

[†] Department of Computers and Systems Engineering, Tokyo Denki University

viewed to move from one location to different locations. *Mobile* objects are objects which can move from one location to others in the system. *Fixed* objects are in the fixed stations. Each object is considered to support some *quality of service* (QoS) like *response time* and *bandwidth*. Thus, according to the movement of the object o, the QoS of o is *changed*. The movement of o is modeled as the *change* of the QoS supported by o in this paper. Problem is how to support users with the service required by the users under situations where the objects are moving in the system. In this paper, we would like to discuss how to manage transactions which manipulate mobile and replicated objects, which support nested, abstract operations.

In Section 2, we present the system model. In Section 3, we discuss how to compute disconnect operations. In Section 4, we present how to compute operations on mobile objects. In Section 5, we discuss how to maintain the mutual consistency among the replicas.

2. System Model

The distributed system is composed of multiple stations interconnected by communication networks (**Fig. 1**). There are two kinds of stations, i.e., *fixed* and *mobile* ones. The fixed stations are connected at the fixed location of the network. The mobile stations communicate with the mobile support station (MSS) by using the wireless channel. If the mobile station moves to another cell, it communicates with the mobile support station in the cell. By the current network technologies[17),18)], the connection among the stations can be maintained while the stations are moving.

A unit of resource in the system is referred to as *object*, which is composed of abstract data and operations for manipulating the data. Each object o can be manipulated only by the operations supported by o. We assume that each object is stored in one station.

There are two kinds of objects, i.e., *class* and *instance*. The class includes the scheme of the data and the operations for manipulating the data. The instance is composed of the data instance of the scheme and the operations inherited from the class.

The objects may be replicated into multiple *replicas* which are in different stations. Here, suppose that an object o is replicated into multiple replicas $o^1, \ldots, o^l$ ($l \geq 2$) where each o^i is in a station s_i ($i = 1, \ldots, l$). If the replicas have the same data and operations as o, o is referred to as *fully replicated* to $o^1, \ldots, o^l$. If not, they are *partially replicated*. First, suppose that o is a class. s_i has all operations supported by o if o is fully replicated. s_i has some operations of o if partially replicated. Next, suppose that o is an instance. Each s_i has the data instance and the operations. If o is partially replicated, s_i has a part of the data of o.

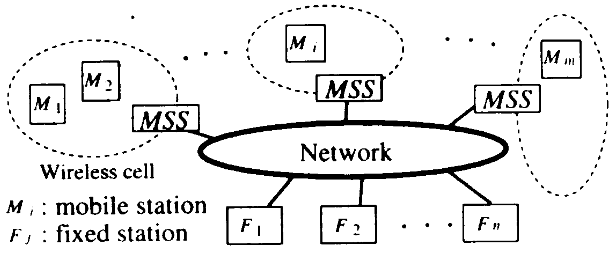

Fig. 1 System model.

If an object o is in a mobile station, the location of o is changed according to the movement of the station. We would like to discuss how the movement of o is viewed. For example, the response time to manipulate o may be increased due to the increased latency to o. Thus, the movement of o is modeled to be the change of the quality of service (QoS) supported by o.

[Definition] An object o is *mobile* iff the *QoS* supported by o is time-variant. □

The computation of an operation *op* in an object o may invoke operations in other objects. The computation of *op* is considered to be *atomic*. That is, all the operations invoked by *op* complete successfully or none of them. If some operation invoked by *op* fails, all the operations invoked by *op* have to be aborted. The computation of each operation invoked by *op* is also atomic. Hence, the computation of the operation is considered to be a *nested* transaction[16),19)].

3. Operations on Disconnected Objects

We would like to discuss how to compute operations on mobile and replicated objects.

3.1 Disconnected Operations

Suppose that there are three objects o_i, o_{ij}, and o_{ijk} with the data d_i, d_{ij}, and d_{ijk}, respectively. Suppose that an operation op_i in o_i invokes an operation op_{ij} in o_{ij} and op_{ij} further invokes op_{ijk} in o_{ijk} as shown in **Fig. 2**. op_{ij} manipulates d_{ij} in o_{ij} and op_{ijk} manipu-

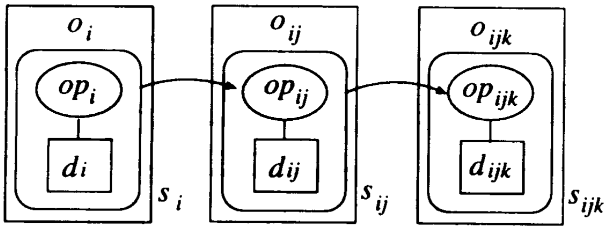

Fig. 2 Invocation.

lates d_{ijk} in o_{ijk}. Since the mobile station is not equipped with such a powerful battery that it can have long-time communication, the mobile station often has to close the connection with other stations to reduce the power consumption. The mobile station also may be disconnected due to jamming and noise. Thus, the operations may be *disconnected*[11] while the operations are being computed. If the object has no connection with the other objects, the object is a *disconnected*. Objects which are not disconnected are *connected*.

There are ways to continue the distributed computation on mobile and fixed objects in the presence of the disconnected objects:

(1) migration of objects, and

(2) replication of objects.

In the migration way, the operations and data of the disconnected object are transferred to another station. On behalf of the disconnected operations, the operations migrated are continued to be computed. The data caching is a kind of migration where only data in the object is copied to another station. In the replication way, the object is replicated into multiple replicas $o^1_{ij} \ldots o^n_{ij}$. If a replica o^k_{ij} used by an object o_i is disconnected, o_i manipulates another replica o^l_{ij} on behalf of o^k_{ij}.

3.2 Migration of Objects

First, we would like to discuss how to migrate objects from one station to others. Here, suppose that o_{ij} in Fig. 2 is to be disconnected due to the close of the connections. There are two ways for migrating the object:

(1) to migrate the disconnected object o_{ij} in s_{ij} to another station, and

(2) to migrate the connected object o_{ijk} in s_{ijk} to the disconnected station s_{ij}.

One way is to migrate the disconnected object o_{ij} to another station. For example, op_{ij} and d_{ij} of o_{ij} are migrated from s_{ij} to another station s_{ijk} as shown in **Fig. 3**. If s_{ijk} has the class of o_{ij}, only d_{ij} can be migrated to s_{ijk} since s_{ijk} has the operation op_{ij}. After migrating o_{ij} to s_{ijk}, op_i can still invoke op_{ij} of o_{ij} in s_{ijk}. If o_{ij} in s_{ij} is reconnected, o_{ij} waits until op_{ij} in s_{ijk} completes. Then, d_{ij} in s_{ijk} is sent to s_{ij} if d_{ij} is changed by op_{ij}. On receipt of d_{ij}, d_{ij} is restored to the data in o_{ij}. In stead of migrating o_{ij} to s_{ijk}, o_{ij} may be migrated to s_i or the other station.

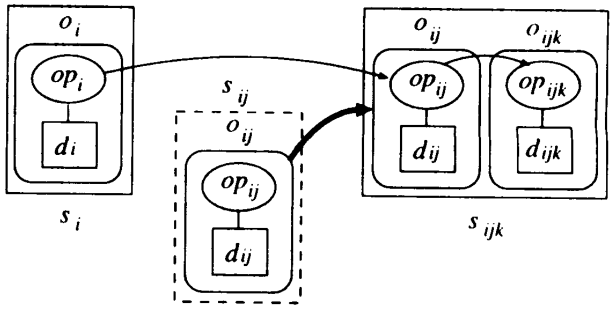

Fig. 3 Migration of o_{ij} to s_{ijk}.

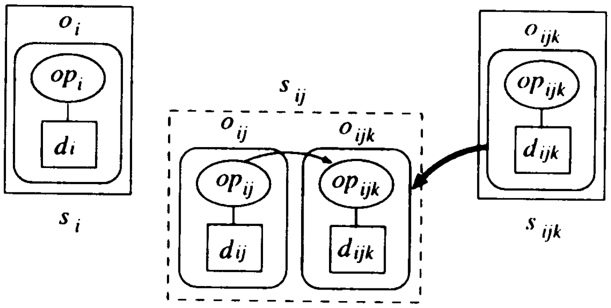

Fig. 4 Migration of o_{ijk} to s_{ij}.

Another way is to move the connected objects to the station s_{ij} to be disconnected. For example, suppose that o_{ijk} is migrated to s_{ij} as shown in **Fig. 4**. o_{ijk} is migrated to s_{ij} from s_{ijk}. Since o_{ijk} is still connected, o_{ijk} is manipulated by other objects while d_{ijk} is being manipulated in s_{ij}. In the caching method, only d_{ij} is sent to s_{ij} assuming that s_{ij} has the class of o_{ij}, i.e., operations for manipulating d_{ij}. It is problem how to maintain the mutual consistency of d_{ijk} among s_{ij} and s_{ijk}. The problem is discussed already by many papers[3),9)].

As stated now, if o_{ij} is to be disconnected, there are two migration ways, i.e., (1) op_{ij} and d_{ij} of o_{ij} in s_{ij} are migrated to another station or (2) op_{ijk} and d_{ijk} invoked by op_{ij} are migrated from s_{ijk}. It depends on which object o_{ij} or o_{ijk} coordinates the distributed computation. For two objects o_{ij} and o_{ijk}, if o_{ij} coordinates the computation on o_{ij} and o_{ijk}, o_{ij} is referred to as *superior* to o_{ijk}. An object which is not superior is migrated to a superior object. For example, if o_{ij} is in the mobile handy sta-

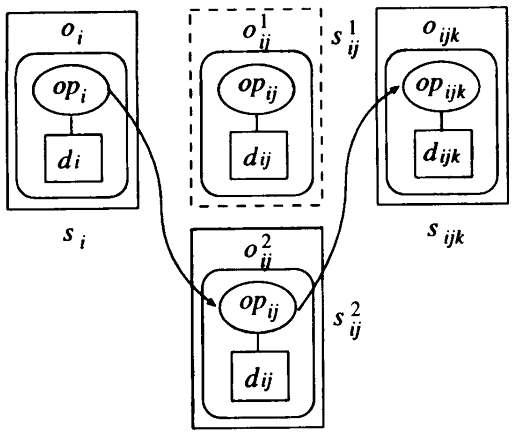

Fig. 5 Replication of o_{ij}.

tion and a user interactively manipulates o_{ijk} through o_{ij}, o_{ij} is superior to o_{ijk}, i.e., o_{ijk} is migrated into s_{ij}. If neither o_{ij} nor o_{ijk} are superior, o_{ij} and o_{ijk} are referred to as *equivalent*.

Suppose that o_{ij} and o_{ijk} are equivalent. The following migration strategy is adopted to reduce the communication overhead:

[Selection of objects]

(1) If either o_{ij} or o_{ijk} is updated, the object whose state is not changed is moved to the other.

(2) If a volume of operation and data to be sent to s_{ijk} is smaller than o_{ij}, o_{ijk} is migrated to s_{ij}. Otherwise, o_{ij} is migrated to s_{ijk}. □

Suppose that an object o_{ij} is updated by the operation op_{ij} and o_{ijk} is not updated by op_{ijk}. If the object o_{ij} is migrated to another station s_{ijk}, o_{ij} in s_{ij} has to be synchronized with the object migrated in s_{ijk} when o_{ij} in s_{ij} is reconnected.

3.3 Replication of Objects

We would like to discuss a case that o_{ij} is replicated into multiple replicas. If one replica o^h_{ij} being manipulated is disconnected, another replica o^k_{ij} is used on behalf of o^h_{ij}. Suppose that o_{ij} is replicated into two replicas o^1_{ij} and o^2_{ij} as shown in **Fig. 5**. In this paper, we assume that the objects are fully replicated, i.e., o^1_{ij} and o^2_{ij} are the same as o_{ij}. If o^1_{ij} is to be disconnected, op_i can invoke op_{ij} in the replica o^2_{ij} and op_{ij} in o^2_{ij} can invoke op_{ijk} as shown in Fig. 5. Here, the current state of o^1_{ij} has to be sent to o^2_{ij}. On receipt of the states of d_{ij} and op_{ij}, the states are restored to d_{ij} and op_{ij} in o^2_{ij} and then o^2_{ij} starts to compute op_{ij} for the current state received from o^1_{ij}.

Another way is to abort op_i. By the abortion of op_i, op_{ij} and op_{ijk} are aborted. Then, op_i is restarted and op_i invokes op_{ij} in o^2_{ij} again.

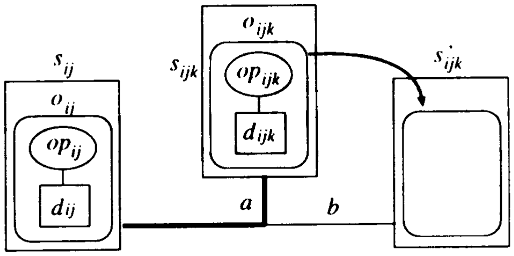

Fig. 6 Movement of object.

4. Operations on Mobile Objects

We would like to discuss how to compute operations on mobile objects.

4.1 Less-qualified Operations

Each object supports some service, i.e., operations for other objects. The quality of service means the performance, reliability, availability, and security aspects of the operations. According to the movement of the object, the *quality of service* (QoS) supported by the object is changed while the object supports the same service. For example, the bandwidth between o_{ij} and o_{ijk} in Fig. 2 is changed to be lower if o_{ijk} is moved to a station s'_{ijk} which is connected with the lower bandwidth network b (**Fig. 6**). Thus, the QoS of o_{ij} is defined for each object o_{ij} issuing operations to o_{ijk}. $QoS(o_{ij}, o_{ijk})$ denotes the QoS which o_{ij} supports for o_{ijk}.

Suppose that o_{ij} is replicated to two replicas o^1_{ij} and o^2_{ij} as shown in Fig. 4. If the QoS of o^1_{ij} for o_i, i.e., $QoS(o_{ij}, o_i)$ is degraded to be lower than op_i expects to take from o_{ij}. o_i can use another replica o^2_{ij} in stead of o^1_{ij} if o^2_{ij} supports the better QoS than o^1_{ij}. Here, let $ReQ(o_{ij}, o_i)$ denote the QoS which o_i requires o_{ij} to support. Let Q_1 and Q_2 denotes two QoS values. $Q_1 \leq Q_2$ means that Q_1 is *better* than Q_2. For example, if Q_1 and Q_2 represent the bandwidths 10Kbps and 1Mbps, respectively, $Q_1 \preceq Q_2$.

[Definition] Suppose that op_i invokes op_{ij}. op_{ij} is referred to as *less-qualified* for op_i if $QoS(o_{ij}, o_i) \preceq ReQ(o_{ij}, o_i)$. □

Suppose that an object o_{ij} is replicated to l (≥ 2) replicas $o^1_{ij}, \ldots, o^l_{ij}$, where each replica o^k_{ij} is stored in a station $s^h_{ij} (h = 1, \ldots, l)$. Let $r(o_{ij})$ be a set of replicas $o^1_{ij}, \ldots, o^l_{ij}$. o_i has to find a replica o^k_{ij} whose QoS is the best in $r(o_{ij})$. o_i selects one replica o^k_{ij} among the replicas in $r(o_{ij})$ as follows.

[Selection of the replica]

(1) o_i sends *Rq-QoS* messages to all the replicas $o_{ij}^1, \ldots, o_{ij}^l$.

(2) On receipt of the *Rq-QoS* message from o_i, each replica o_{ij}^k sends back the *Rp-QoS* message with the QoS of o_{ij}^k to o_i $(k = 1, \ldots, l)$.

(3) If o_i receives the *Rp-QoS* messages from the replicas, o_i selects one replica o_{ij}^k with the best QoS among them. □

Then, op_i invokes the operation op_{ij} in o_{ij}^k. This method implies larger communication overhead to broadcast *Rq-QoS* messages to all the replicas. Hence, we adopt the following heuristics to select the replica.

[Selection of the replica]

(1) If there is a replica o_{ij}^k in the same cell as o_i, the replica $o_{ij}^k$is selected. If there are multiple replicas in the cell, the replica o_{ij}^k which supports o_i with the best QoS among them is selected.

(2) If there is no replica in the cell, o_i broadcasts the *Rq-QoS* message by the selection algorithm. □

Another way is that there is one coordinator of the replicas, say o_{ij}^1. o_{ij}^1 monitors the change of QoS of each replica o_{ij}^k. o_i first asks o_{ij}^1 to find the best replica for o_i. Then, o_{ij}^1 selects the best one, say o_{ij}^k.

While op_{ij} is computed in o_{ij}^k, the QoS of o_{ij}^k may be changed according to the movement of o_{ij}^k. If op_{ij} could not support the QoS required, i.e., op_{ij} is less-qualified, o_i can select another replica of o_{ij} which supports the better QoS than o_{ij}^k.

[Resolution of the replicas]

(1) If the QoS of o_{ij}^k is being degraded for some time units, o_i finds the best replica o_{ij}^h which is better than o_{ij}^k by the selection procedure.

(2) If o_{ij}^h is detected, o_i requires o_{ij}^k to send the states of d_{ij} and op_{ij} to o_{ij}^h. On receipt of them from o_{ij}^k, o_{ij}^h restores them to the state. o_i invokes op_{ij} in o_{ij}^h. □

4.2 Faulty replicas

One problem on considering the disconnected operations is how to differentiate disconnected objects from faulty objects. Suppose that o_{ij} is faulty in Fig. 2. If o_{ij} stops by failure, the connection with o_{ijk} is closed. o_{ijk} cannot know whether o_{ij} is faulty or not because the connection is closed and there is no way to communicate with o_{ij}. Here, we make the following assumptions:

[Assumptions]

(1) The network is synchronous, i.e., the propagation delay is bounded.

(2) The computations in the objects are synchronous[6]. □

The assumptions mean that faulty objects can be detected by the timeout mechanism.

We adopt the following strategy to detect faulty objects:

[Detection of faulty objects]

(1) The disconnected object o_{ij} sends periodically an *Alive* massage to o_i and o_{ijk}.

(2) After the disconnection, if o_{ijk} or o_i does not receive any message from o_{ij} for some predetermined time units, o_{ijk} considers that o_{ij} is faulty. □

The operational objects have to send *Alive* messages to inform other objects of their being operational. The *Alive* message is sent by using the connectionless communication.

4.3 Computation of QoS

Since o manipulates o_i by an operation op_i, the QoS is redefine as $QoS(o_i{:}op_j, o)$. We would like to discuss how $QoS(o_i{:}op_j, o)$ is computed. Since operations in objects are nested, $QoS(o_i{:}op_j, o)$ depends on not only the computation of actions in o but also QoSs of operations invoked by op_i. Suppose that op_i invokes operations $op_{i1}, \ldots, op_{im}$ of objects $o_{i1}, \ldots, o_{im}$, respectively. $QoS(o_i{:}op_i, o)$ is computed as follows:

$$QoS(o_i : op_j, o) = f_i(QoS(o_{i1} : op_{i1}, o_i), \\ \ldots,\ QoS(o_{im} : op_{im}, o_i), qos(op_i, o)).$$

Here, $qos(op_i, o)$ denotes the QoS required for op_i to manipulate o_i. f_i is a function which gives the QoS of op_i from the QoSs of $op_{i1}, \ldots, op_{im}$. There are kinds of QoSs. The computation time of op_i is obtained by adding the computation times of $op_{i1}, \ldots, op_{im}$ and op_i, i.e., f_i is "+" if op_i is computed sequentially. If $op_{i1}, \ldots, op_{im}$ in op_i are computed in parallel, the QoS is obtained by taking the maximum one among of $op_{i1}, \ldots, op_{im}$.

In order to compute the QoS of op_i, o_i asks o_{ij} to send $QoS(o_{ij}{:}op_{ij}, o_i)$ periodically or each time op_i is invoked.

5. Type Based Optimistic Concurrency Control

We would like to discuss how to maintain mutual consistency among the replicas.

5.1 Lock Modes

Objects may be replicated. Here, for an object o_i, let $r(o_i)$ be a collection of replicas of o_i, i.e., $r(o_i) = \{o_i^1, \ldots, o_i^{l_i}\}$ $(l_i \geq 2)$, where each o_i^j is a replica of o_i $(j = 1, \ldots, l_i)$. Each replica o_i^j is stored in a station s_{ij} $(j = 1, \ldots, l_i)$. We would like to discuss how to maintain mutual consistency among the replicas.

Before an operation op_i is applied to o_i, o_i is locked. If o_i is locked, op_i is computed in o_i. If not, op_i waits. Two operations op_i and op_j are referred to as *compatible* iff the states obtained by computing op_i and op_j in any order are the same. In order to increase the concurrency, kinds of lock modes are introduced, e.g., *read* and *write* modes. The objects support more kinds of operations than *read* and *write* of the file objects. An operation op_i of o_i is assigned a lock mode $m(op_i)$. The compatibility relation among the lock modes is defined as follows[13].

[Definition] For every pair of lock modes m_1 and m_2 supported by an object o_i, m_1 is *compatible with* m_2 iff an operation of m_1 is compatible with operations of m_2. □

If m_1 is not compatible with m_2, m_1 *conflicts* with m_2. That is, op_i of m_1 has to wait until the operations of m_2 complete in o_i. For example, a *Bank* object supports operations *deposit* and *withdrawal*. The modes of *deposit* and *withdrawal* are compatible.

Objects support various kinds of abstract operations like *deposit* and *withdrawal* while the database systems support only *read* and *write* operations. Hence, various kinds of lock modes are supported by the objects. The precedence relation among the lock modes is formally defined by Korth[13]. Here, let M_O be a set of lock modes supported by an object o. For each mode m in M_O, let $c(m)$ $(\subseteq M_O)$ be a set of modes which m is compatible with.

[Definition] For every pair of modes m_1 and m_2 of an object o, $m_1 \prec m_2$ (m_2 is *stronger* than m_1) iff $c(m_1) \supseteq c(m_2)$. □

Here, $m_1 \preceq m_2$ means that m_2 is stronger than m_1. If neither $m_1 \prec m_2$ nor $m_2 \prec m_1$, m_1 and m_2 are *equivalent* ($m_1 \parallel m_2$). Here, $m_1 \preceq m_2$ or $m_1 \parallel m_2$. Here, $read \prec write$ because $c(read) = \{read\} \supseteq c(write) = \phi$.

5.2 Optimistic Locking

The typical scheme to maintain the mutual consistency among multiple replicas is the *read-one* and *write-all* (ROWA) principle. That is, the read operation is issued to one replica while the write operation is issued to all the replicas. If one replica is locked in a read mode, the read operation can be computed in the replica. On the other hand, if all the replicas could be locked in the *write* mode, the write operation is computed in all the replicas. In order to reduce the communication overhead, the optimistic approach[14] is adopted. Carey[4] discusses the optimistic two-phase locking (O2PL) protocol. Jing[11] extends the O2PL so as to reduce the communication overhead by avoiding the releases of the locks. In the O2PL, one replica is locked by *read* but the replicas are not locked by *write*. When the transaction commits, the replicas updated are locked by *write*. More abstract types of operations are considered in the objects than the *read* and *write* operations. The read-one and write-all principle can be extended by taking into account the various kind of lock modes.

The second point on the operations is concerned with whether the operations change the state of the object or not. For example, *deposit* and *withdrawal* change the state of *Bank* while they are compatible. If an operation *op* does not change the state of o, *op* can be computed in only one replica of o. Otherwise, *op* has to be computed in all the replicas to keep the mutual consistency among the replicas.

The third point is concerned with whether the operations invoke another operation or not. Suppose that an operation op_i in o_i invokes op_{ij} in o_{ij} and o_i is replicated to two replicas o_i^1 and o_i^2. If op_i is computed in o_i^1 and o_i^2, op_{ij} is invoked twice, i.e., by op_i in o_i^1 and o_i^2. It implies the inconsistency among o_i and o_{ij}. Hence, if an operation in an object invokes another operation, the operation can be computed in only one replica and the state obtained by computing the operation in the replica has to be transferred to the other replicas to make the states consistent.

[Optimistic locking] Suppose that an operation *op* of a mode m_1 is issued to o.

(1) If $m_1 \prec m_2$ for every mode m_2 of o, one replica o^k in $r(o)$ is locked, and *op* is computed in o^k if *op* does not change the state

of o, otherwise op is computed in all the replicas,

(2) Otherwise, all the replicas in $r(o)$ are locked, and op is computed in all the replicas. □

Problem is the communication overhead since all the replicas have to be locked by the operations whose modes are not minimal.

5.3 Optimistic type-based locking

We adopt the optimistic approach to reduce the communication overhead, named *optimistic type-based locking* (*OTL*). We make the following assumption.

[Assumption] The less restricted the operations are, the more often they are used. □

Each operation op locks some number of replicas in $r(o)$ rather than locking all the replicas. The more restricted the operation mode is, the more replicas are locked. For each operation op_i in o_i, a number $q(op_i)$ is given as follows.

- $q(op_i) < q(op_j)$ if $m(op_i) \prec m(op_j)$.
- $1 \leq q(op_i) \leq l_i$.
- for every op_j, if $m(op_i) \preceq m(op_j)$, $q(op_i) = 1$.

op_i locks $q(op_i)$ replicas of o_i. For example, suppose that there are five replicas of an object o_i and o_i has three operations op_{i1}, op_{i2}, and op_{i3}. Suppose that $m(op_{i1}) \prec m(op_{i2}) \prec m(op_{i3})$. $q(op_{i1}) = 1$. $q(op_{i2})$ and $q(op_{i3})$ are, for example, given as 2 and 3, respectively. Before computing op_{i2}, two replicas in five ones are locked.

An operation op_i locks an object o_i by the following scheme.

[Locking scheme]

(1) Before computing op_i, $q(op_i)$ replicas in $r(o_i)$ are locked in a mode $m(op_i)$. Here, let $s(op_i)$ be a subset of replicas in $r(o_i)$ which are to be locked here.

(2) If all replicas in $s(op_i)$ are locked, op_i is computed.

  (a) If op_i invokes operations in other objects, op_i is computed in one replica in $s(op_i)$.

  (b) Otherwise, op_i is computed in all the replicas.

(3) If some replica in $s(op_i)$ is not locked, op_i aborts. □

Since a stronger operation op locks more replicas, op is more often aborted if other stronger operations are manipulating the replicas. If a replica is in the same cell as an object invoking op_i, the replica is locked at step (1). The replicas with the better QoS are selected to be locked as discussed in the preceding subsection.

We would like to discuss how an operation op invoking op_i commits. The commitment of op_i on multiple replicas is coordinated by the two-phase commitment$^{2)}$. One replica o_i^k in $s(op_i)$ plays a role of the coordinator and the other replicas are the participants.

[Commitment]

(1) o_i^k sends a *Prepare* message to all the replicas. The participant replica o_i^j which is not locked by op_i, i.e., o_i^j in $r(o_i) - s(op_i)$, is locked in the mode $m(op_i)$ on receipt of the *Prepare* message. If locked, the replica o_i^j sends back *Yes* message to o_i^k.

(2) If some replica o_i^j in $r(o_i) - s(op_i)$ is not locked, o_i^j sends *No* to o_i^k.

(3) If o_i^k receives *Yes* from all the participant replicas, o_i^k sends *Commit* to all the participants. If o_i^k receives *No* from some participant, o_i^k sends *Abort* to the participants sending *Yes*.

(4) If the participant replica o_i^j receives *Abort*, o_i^j abort op_i if o_i^j had computed op_i.

(5) If the participant replica o_i^j receives *Commit*, all the replicas in $r(o_i) - s(op_i)$ are locked.

  (a) Unless op_i invokes operations in other objects, op_i is computed in all replicas in $r(o_i)$ if op_i changes the state, otherwise op_i commits.

  (b) Otherwise, the state of the replica whose op_i is computed is sent to all the replicas. □

If op_i invokes an operation op_{ij} in another object and op_i is computed in o_i, op_{ij} is computed more than once. In order to avoid the iterated computation, op_i is computed in only one replica, say o_i. In stead of computing op_i in the other replica, the state of o_i is sent to all the other replicas of o_i. If op_i commits, all the locks on the replicas are released.

6. Evaluation

We would like to evaluate the optimistic type-based (OTL) locking scheme by comparing with the traditional read-one and write-all (ROWA) scheme in terms of the number of transactions aborted and the number of lock requests. Here, let o be an object supporting operations $op_1, \ldots, op_h$, which is replicated into replicas

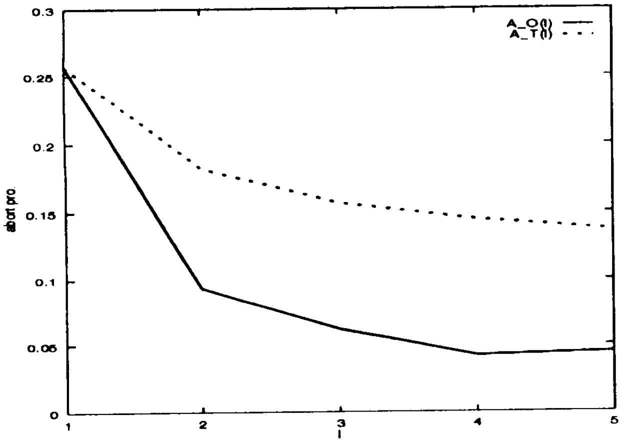

Fig. 7 Probability of abort.

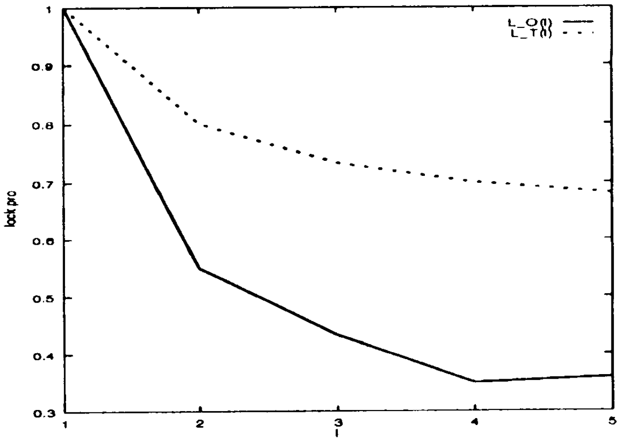

Fig. 8 Probability of lock.

$o^1,\ldots,o^l$, i.e., $r(o) = \{o^1,\ldots,o^l\}$. Suppose that $m(op_i) \prec m(op_j)$ $(i < j)$. Let $f(op_i)$ be the probability that op_i is issued to o. Here, $f(op_1) + \ldots + f(op_h) = 1$. $q(op_i)$ denotes the number of replicas to be locked by op_i in the OTL scheme. Here, $q(op_1) = 1$, $q(op_h) = l$, and $q(op_i) \leq q(op_j)$ if $i < j$, i.e., $m(op_i) \prec m(op_j)$.

Suppose that the operations $op_1,\ldots,op_h$ are randomly issued to o. Let A_O be the probability that an operation is aborted in the OTL scheme. If at least one kind of operation sends the lock request to one replica, the operations are aborted. Hence, A_O is given as $1 - \prod_{i=1}^{h}(1 - f(op_i) \cdot q(op_i)/l)$ $- \sum_{i=1}^{h}[f(op_i)q(op_i)/l \prod_{j=1(j\neq i)}^{h}(1 - f(op_j) \cdot q(op_j)/l)]$. In the OTL scheme, op_i locks $q(op_i)$ replicas. Let A_T be the probability that an operation is aborted in the traditional ROWA way. A_T is obtained by assigning $q(op_1)$ with 1 and $q(op_j)$ with l $(j \geq 2)$ in A_O because only op_1 locks one replica and the other operations lock all the replicas in $r(o)$.

Next, let us consider how many lock requests are sent to the replicas. Let L_O and L_T denote the probabilities that each replica is locked in the OTL and traditional ROWA schemes, respectively. L_O is given by $\sum_{i=1}^{h} f(op_i)q(op_i)/l$. L_T is given by $f(op_1)/l + (f(op_2) + \ldots + f(op_k)) = 1 - f(op_1)(l-1)/l$.

A_0, A_T, L_0, and L_T are computed for the number l of replicas where $h = 5$, i.e., o has five operations. Here, $q(op_i) = \lceil l/2^{h-i} \rceil (i = 1,\ldots,h)$, $f(op_1) = 0.4, f(op_2) = 0.2, f(op_3) = 0.2, f(op_4) = 0.1, f(op_5) = 0.1$. **Figure 7** and **Fig. 8** show A_O and A_T, and L_O and L_T for the number l of replicas, respectively. These figures show that less number of transactions are aborted and less number of lock requests are issued in the OTL scheme than the traditional ROWA.

7. Concluding Remarks

In this paper, we have discussed how to support nested transactions manipulating replicated and mobile objects in the distributed system. We have modeled the mobile objects to be ones whose QoS is changed according to the movement of the objects. We have discussed the optimistic two-phase locking to maintain the mutual consistency among the replicas. Here, the read-one and write-all principal is extended so that the objects can support more kinds of abstract operations than *read* and *write* and the operations are nested.

References

1) Badrinath, B.R., Acharya, A. and Imielinski, T.: Structuring Distributed Algorithms for Mobile Hosts, *Proc. the 14th ICDCS*, pp.21–28 (1994).

2) Bernstein, P.A., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).

3) Barbara, D. and Imielinski, T.: Sleepers and Workaholics: Caching Strategies in Mobile Environments, *Proc. the ACM SIGMOD*, pp.1–12 (1994).

4) Carey, J.M. and Livny, M.: Conflict Detection Tradeoffs for Replicated Data, *ACM TODS*, Vol.16, No.4, pp.703–746 (1991).

5) Clifton, C., Garcia-Molina, H. and Bloom, D.: HyperFile: A Data and Query Model for Documents, *VLDB Journal*, Vol.4, No.1, pp.45–86 (1955).

6) Fischer, J.M., Nancy, A.L. and Michael, S.P.: Impossibility of Distributed Consensus with

One Faulty Process, *Journal of ACM*, Vol.32, No.2, pp.374-382 (1985).

7) Gray, J.: The Transaction Concept: Virtues and Limitations, *Proc. VLDB*, pp.144-154 (1981).

8) Gruber, R., Kaashoek, F., Liskov, B. and Shrira, L.: Disconnected Operation in the Thor Object-Oriented Database System, *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, pp.51-56 (1994).

9) Huang, Y., Sistla, P. and Wolfson, O.: Data Replication for Mobile Computers, *Proc. the ACM SIGMOD*, pp.13-24 (1994).

10) Imielinski, T., Viswanathan, S. and Badrinath, B.R.: Energy Efficient Indexing on Air, *Proc. the ACM SIGMOD*, pp.25-36 (1994).

11) Jing, J., Bukhres, O. and Elmagarmid, A.: Distributed Lock Management for Mobile Transactions, *Proc. the 15th ICDCS*, pp.118-125 (1995).

12) Kistler, J.J. and Satyanaranyanan, M.: Disconnected Operation in the Coda File System, *ACM Trans. Database Syst.*, Vol.10, No.1, pp.2-25 (1992).

13) Korth, H.F.: Locking Primitives in a Database System, *JACM*, Vol.30, No.1, pp.55-79 (1983).

14) Kung, H.T. and Robinson, J.T.: On Optimistic Methods for Concurrency Control, *ACM Trans. Database Syst.*, Vol.6, No.2, pp.213-226 (1981).

15) Lu, Q. and Satyanaranyanan, M.: Isolation-Only Transactions for Mobile Computing, *ACM Operating Systems Review*, Vol.28, No.2, pp.81-87 (1994).

16) Moss, J.E.: Nested Transactions: An Approach to Reliable Distributed Computing, *The MIT Press Series in Information Systems* (1985).

17) Tanaka, R. and Tsukamoto, M.: A CLNP-based Protocol for Mobile End Systems within an Area, *Proc. IEEE International Conf. on Network Protocols (ICNP)*, pp.64-71 (1993).

18) Teraoka, F., Yokote, Y. and Tokoro, M.: A Network Architecture Providing Host Migration Transparency, *Proc. ACM SIGCOM*, pp.209-220 (1991).

19) Weikum, G. and Schek, H.J.: Concepts and Applications of Multilevel Transaction and Open Nested Transactions, *Database Transaction Models for Advanced Applications*, pp.516-553 (1992).

20) Yasuzawa, S. and Takizawa, M.: Uncompensatable Deadlock in Distributed Object-Oriented Systems, *Proc.the International Conf.on Parallel and Distributed Systems (ICPADS)*, pp.150-157 (1992).

21) Yeo, L.H. and Zaslavsky, A.: Submission of Transactions from Mobile Workstations in Cooperative Multidatabase Processing Environment, *Proc. the 14th ICDCS*, pp.372-379 (1994).

(Received October 2, 1995)
(Accepted May 10, 1996)

Takeaki Yoshida was born in Tokyo, Japan, on November 14, 1971. He received his B.E. degree from Dept. of Computer and Systems Enginering, Tokyo Denki University in 1995. He is now a graduate student of the master course in Dept. of Computer and Systems Enginering, Tokyo Denki University. His research interest includes mobile database systems, distributed systems, and computer networks.

Makoto Takizawa was born in 1950. He received his B.E. and M.E. degrees in Applied Physics from Tohoku University, in 1973 and 1975, respectively. He received his D.E. in Computer Science from Tohoku University in 1983. From 1975 to 1986, he worked for Japan Information Processing Developing Center (JIPDEC) supported by the MITI. He is currently a Professor of the Department of Computers and Systems Engineering, Tokyo Denki University since 1986. From 1989 to 1990, he was a visiting professor of the GMD-IPSI, Germany. He is also a regular visiting professor of Keele university, England since 1990. He was a vice-chair of IEEE ICDCS, 1994 and serves on the program committees of many international conferences. His research interest includes communication protocols, group communication, distributed database systems, transaction management, and groupware. He is a member of IEEE, ACM, IPSJ, and IEICE.