Object-Oriented Transaction Support for Distributed Persistent Objects

Toshio Hirotsu † Mario Tokoro ‡

Department of Computer Science,
Keio University
3-14-1 Hiyoshi, Yokohama 223 Japan

Abstract

We propose a scheme for guaranteeing the consistency of persistent objects in a distributed environment. Since persistent objects are referenced and updated concurrently by many programs, a distributed system should support some transaction mechanism. We propose a transaction scheme based on Serialization Graph Testing, which preserves consistency of persistent objects without using global information. A Transaction Management Object (TMO) is created for each transaction, and consistency is preserved by negotiation among TMOs. Since we employ a non-locking algorithm and assume no global entity, our scheme can manage transactions efficiently even in a very large distributed environment. Furthermore, TMOs can reflect the characteristics of transactions and the objects involved to the transaction management. Thus, flexible transaction management is achieved.

1 Introduction

In distributed systems, many people work at different spatial and temporal locations. To share data among users at different temporal locations, persistent objects are used. When objects are accessed concurrently, it is an important issue to decide in what conditions the effects of the operations on the objects are to be reflected in their persistent[1]states. An inconsistent update on an object should not be reflected in its persistent state, since it may be referred from other users. Therefore, consistent updates and references to shared objects become an essential requirement. They can be guaranteed by introducing some transaction management mechanism. Several systems such as Argus [5], Camelot/Avalon [13, 3], Eden [1] and Arjuna [12] incorporate some kind of transaction facility.

When considering the sharing of resources, the notion of the object is attractive. An object is a unique abstraction unit of sharing, of resources, and of replication against failures. Owing to the modularity of capabilities, such as the mechanisms of deciding the orders among transactions, different forms of capabilities can easily be selected. Our research goal is to construct such a computing system in which each distributed resource is represented as an object, each resource management mechanism is realized by objects, and tasks in the system are processed through communication among objects. We call such a system *DOOS* (Distributed Object-Oriented System). In the lowest layer of *DOOS*, an object-oriented operating system such as Apertos[2] [19, 18, 17] is assumed, and all the facilities of the higher layers are to be implemented on top of it.

In distributed systems, users usually work in different contexts and levels, and towards different goals. Therefore, various forms of resource sharing have to be considered. Moreover, since the configuration of the system is changing dynamically and frequently, flexible and dynamic transaction management mechanisms have to be supported. In order for the transaction management mechanism to be used in a very large distributed environment, it should not depend upon any global entity. In addition, the transaction management mechanism should not be based on locking, since locking may impede concurrency.

In this paper we propose a new object-oriented transaction management mechanism supporting distributed persistent objects in very large distributed

† *hirotsu@mt.cs.keio.ac.jp*

‡ *mario@mt.cs.keio.ac.jp*, Also affiliated with Sony Computer Science Laboratory Inc., 3-14-13 Higashi Gotanda, Shinagawa-ku, Tokyo 141, Japan. e-mail:*mario@csl.sony.co.jp*

[1] i.e., *permanent* in the database terminology.

[2] Formerly called the Muse Operating System.

systems. This scheme is more efficient and flexible than ones based on local atomicity[14] and cooperative atomicity [9]. Our scheme is based on Serialization Graph Testing [2], a non-locking scheme, and we introduce the Transaction Management Object (*TMO*) which is created for each transaction. Each object has a facility for storing its own state as its persistent state. Possible inconsistency on a shared object is reported to the related *TMO*, and consistency is achieved by negotiation among *TMO*s. Since we assume no global entity, our scheme can manage transactions efficiently even in a very large distributed system. In addition, *TMO*s can reflect the characteristics of transactions and that of objects involved in negotiation. For example, *TMO*s can negotiate the selection of victims to reduce the cost of abortion.

The organization of the paper is as follows: Section 2 deals with the transaction facilities required in object-oriented distributed systems. In section 3 we define our object and transaction model. Next, in section 4 we briefly present algorithms for consistent accesses to objects. Section 5 discusses the current implementation of our scheme. In section 6, we compare our scheme with related works. We conclude with our achievements and summarize our future work in section 7.

2 Transactions in Distributed Systems

This section describes the requirements for a transaction management system in Distributed Object-Oriented Systems(*DOOS*). First, we briefly describe existing transaction systems, and then discuss transactions for distributed persistent objects.

2.1 Transactions in Existing Systems

Transaction mechanisms have been used in the field of databases for consistent updates of shared data in a multi-user environment [2]. A transaction is a sequence of operations that refer to or update data. In order to guarantee consistency for concurrent accesses, each transaction is executed atomically. This facility is called atomicity, and it is guaranteed by serializability and recoverability. Transaction mechanisms used in databases are usually constructed based on Read/Write operations on data. Techniques such as Two Phase Locking, Time Stamp Ordering, and Multiversion mechanisms [2] have been used to ensure atomicity.

There are also studies which incorporate transaction facilities in programming languages and operating systems. Argus [5, 6], Camelot/Avalon [13, 3], and ERMS [11] are such examples. Argus is a programming language developed at MIT for distributed programming incorporating atomic actions to enable programmers to deal with failures. Resources are encapsulated into a Guardian (a kind of abstract object), and resources in a Guardian can only be accessed through handlers (special procedure to manipulate resources in the Guardian). Since calls to handlers can be made from other calls (nested calls), all calls are implemented as actions. Camelot/Avalon is a system developed at CMU; Camelot supports various functions for transaction processing on the Mach operating system. Avalon runs on Camelot and supports facilities for programming transaction-based computations at the language level. ERMS (Eden Resource Management System) is a transaction management mechanism introduced in Eden [1]. The ERMS Transaction Manager is composed hierarchically; at each level, ERMS uses known concurrency control and crash recovery mechanisms which were developed for single-level transaction systems. The fundamental building block of ERMS is the ERMS Transaction Manager (ETM) which handles concurrency control and crash recovery of a transaction. Each ETM manages a lock table and a map in order to keep track of committed versions. At the system level, a System Lock Manager serializes top-level transactions and a system directory called R2D2 stores the authoritative versions of resources.

There are other notions based on commutativity relations between operations: local atomicity and cooperative atomicity. A scheme based on local atomicity property was proposed by Weihl [14], where all objects are passive and do not exchange information about transactions; objects maintain atomicity by themselves. When using this scheme, the serial order is determined by a global clock or the order of the commit operations. For instance, a two phase locking mechanism based on commutativity of operations is described in [15]. In this scheme, a lock is obtained based on commutativity of operations, when results from operations are returned. Two operations which are not commutative conflict. If a lock cannot be obtained, the corresponding operation is delayed. By using local commutativity-based locking, when a conflict occurs, one transaction is delayed until the other transaction is committed; the delayed time is unpredictable, and therefore response time is sacrificed. Furthermore, by using a global clock, the delay for collecting global information increases as the system gets larger.

In the scheme based on cooperative atomicity [9], objects are active agents which exchange their dependency relations with others in order to decide the serial order of transactions. The scheme tries to decide serial order only between conflicting transactions; thus no global information is necessary, and the cost for deciding serial orderings is not expensive even for large systems. Moreover, the system does not need to wait for commitment of conflicting transactions before executing the next transaction; it only needs to ensure that objects are consistent. Hence, a better response time can be achieved when compared to the scheme based on local atomicity. However, according to cost considerations of each transaction, there may be other transactions more appropriate to be aborted . In addition, all objects related to a transaction exchange messages to detect the inconsistency in the transaction. Moreover, since any object can be a coordinator of objects at conflict time, all objects must be provided with the entire mechanism for consistency. Consequently, the policy of the transaction management and the mechanism for consistency cannot be clearly separated.

2.2 Transaction Mechanisms for Distributed Objects

When we change the state of a shared object, it is necessary to ensure the consistency among objects, before the new state is reflected to the persistent state. Two kinds of consistencies can be considered.

1. Intra-object consistency : When multiple messages arrive at one object simultaneously (at the same time or while executing another method), updating the object should be done concurrently while keeping its consistency.

2. Inter-object consistency : If the states of several objects are changed, consistency of states among all these objects have to be preserved.

For intra-object consistency, the execution of each method in an object must be atomic. In concurrent object-oriented languages such as Concurrent-Smalltalk[16], or Actor based languages such as Acore[7], atomicity in method execution is preserved by message queues or serializers which serialize incoming messages and enforce the execution of a method only after the completion of the previous one. Using this technique, every operation blocks other operations. Therefore, even non-conflicting operations that could be executed concurrently without problems (like read/read) are blocked. A solution to this problem is to use a concurrency control mechanism for each method executing in an object.

For inter-objects consistency, execution order of transactions must be the same in all the objects involved in a transaction. A transaction in an object-oriented system is considered as a sequence of method executions in objects which are not interleaved by other method executions. In Argus [6] and Arjuna [12, 10] a locking mechanism is used to realize this facility.

Now let us consider the characteristics of transactions in *DOOS*. There are many types of transactions; their characteristics are defined by the transactions' lifetime, number of messages sent from other transactions to objects in the transaction, degree of contention in executing the transaction, the importance of the transaction, and so on. Thus, we would like to abort the most suitable transaction when inconsistent state has occurred. For example, a transaction that has executed for long time or which many transactions depend on may be expensive to abort.

Furthermore, we cannot assume global entities in distributed systems. Because of this, choosing victims to resolve inconsistency and the time at which an object's state is to be reflected to the persistent state must be decided locally. A local management mechanism is desirable for these features.

3 The Model of Transaction

So far, we have considered the transaction mechanisms required in *DOOS*. In this section, we consider a model of objects, transactions, and computations that makes the local management mechanism feasible, and that is suitable for a distributed system.

3.1 Objects and Transactions

An object is constructed from resources (data) and operations encapsulating them. Operations are also called methods. Computation takes place by sending messages to an object and invoking the corresponding method in the object. If each object has all information about its own actions and knows what to do when receiving a message, then the object can decide the action to be taken by itself. Note that objects can refer to or modify its own state (internal actions) and can send messages to other external objects (external actions), as depicted in Figure 1. Each object has capability to make it persistent by writing its state to persistent storage, but it doesn't know when it should

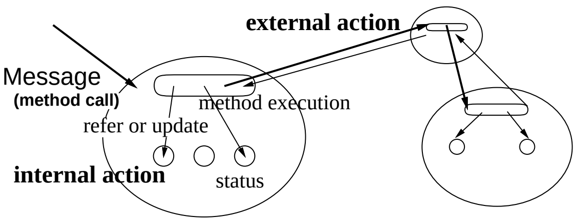

Figure 1: Object-Oriented Model

be done. In our model, an object does not execute multiple methods concurrently; in other words, there is no concurrency within an object. Owing to this limitation, the order of two methods executed in an object is clearly decided. This limitation can be removed by using Linearizability [4], for concurrency is allowed within an object if it can be observed as sequential.

A transaction is a sequence of messages that are sent
between a start_transaction and an end_transaction. An object having method containing start_transaction and end_transaction initiates the transaction. Since message may activate other messages, objects related to a transaction are not only the ones that receive the message from initiator of the transaction but also all the subsequently called objects. The execution of a transaction is depicted in Figure 2. In our model, we do not allow concurrency within a transaction. Concurrency in a transaction is caused by asynchronous message send; this limitation can be removed by using Nested Transactions [8].

When two methods are executed in an object, there may be dependency between them; this is called conflict of methods. Transactions which execute conflicting methods are called conflicting transactions. Conflicting transactions have dependencies (order relation) on their execution, and if a loop occurs, those transactions are in an inconsistent state. Considering about commutativity of the methods [14], occurrence of such a conflict and therefore inconsistency can be reduced.

3.2 Transaction Set

An object can be shared: it may receive messages from several objects. Note that the objects that may be shared and those that may cause transactions to conflict cannot be statically predicted. This means that the set of objects which conflict with each other

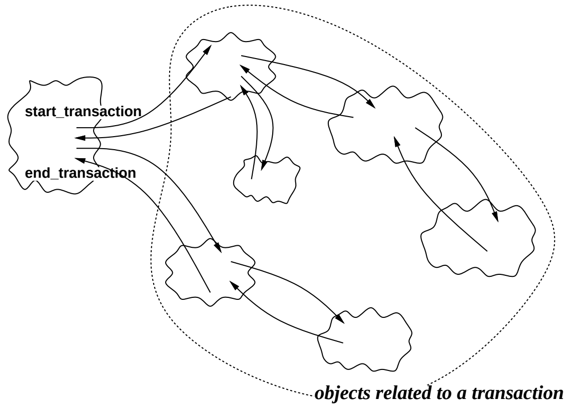

Figure 2: Transaction among Objects

can only be determined at run-time.

Let us now consider conflicts between transactions. A conflict between transactions occurs when

- there is an overlap in the sets of objects which receive messages from different transactions, and
- messages sent to objects within the overlapped set conflict with each other.

From this consideration, in order to manage transactions, we require that all objects that receive messages in a transaction are managed together. Some terms are defined as follows:

Transaction Set

Transaction Set is defined as the set of all objects related to a transaction. That is to say, the objects which receive messages between a start_transaction and an end_transaction compose a *Transaction Set*. Let the *Transaction Set* of transaction T_i be TS_i.

Shared Set

Shared Set is defined as the set of objects that compose the overlapped part of *Transaction Sets*. The *Shared Set* S_i of transaction T_i is

$$S_i = \bigcup_{j}^{n} (TS_i \cap TS_j) \qquad (\text{for } i \neq j)$$

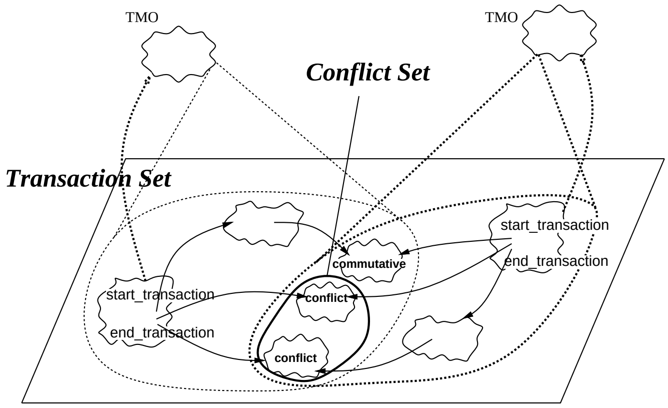

Figure 3: *Transaction Set and TMO*

Conflict Set

> *Conflict Set* is defined as the subset of the *Shared Set* on which conflict operations are actually invoked. Objects in this set may cause inconsistency.

In the model of objects and transactions described above, in order to manage consistency of references and updates between transactions, conflicts of transactions are detected within each object, and inconsistency of transactions is resolved by controlling the order of method execution so that it does not interleave with other objects in the *Conflict Set*.

3.3 Transaction Management Object

Let us now consider how a transaction management mechanism can be represented. The primary requirements for our transaction mechanism are:

- No global entity should be employed. That is to say, global transaction manager, global lock manager, global data manager etc. cannot be used.
- The transaction management scheme should provide fine granularity for controlling transactions in order to provide higher flexibility.

Therefore, in our model of transactions, a single object that manages the information of a transaction (*Transaction Set*, *Conflict Set*, life-time, etc.) is created for each transaction; it is called *Transaction Management Object*(*TMO*). Inconsistent states between transactions can be detected using the order relation of conflicting transaction that each *TMO* has, and can be resolved through communication among *TMO*s (Figure 3).

The outline of the transaction management scheme using *TMOs* is as follows:

- A *TMO* is generated by the execution of a `start_transaction` command in a method.
- The identifier of that *TMO* is passed to all the objects accessed in the transaction (the elements of the *Transaction Set*). Messages sent in a transaction contain the *access history*; each object accessed in the transaction appends its identifier to the access history.
- When the method call returns, the *access history* is sent along with the reply messages.
- When the method call returns to the initiator of the transaction the *access history* is passed to the *TMO* and the *Transaction Set* is constructed.
- When a conflict between methods occurs in an object, the *access history* is sent the *TMO*s whose transactions execute the conflicting methods. The *TMO*s append the object to its *Conflict Set*, and append the dependency of conflicting transactions to its local information on order relation.
- *TMO*s check the dependency of transactions from the order relations which each *TMO* possesses, in order to detect the inconsistencies.
- When an inconsistent state is detected, conflicting *TMO*s negotiate with each other to resolve the inconsistency.

- From the result of the negotiation, a serial order is decided, appropriate transactions are selected for victims according to the characteristics of the transaction, and consistency between transactions is preserved.

- When the execution of a transaction reaches the end_transaction, the initiator of the transaction notifies the *TMO* that it can commit the transaction.

- *TMO* checks that no transaction that should be committed or aborted prior to the committing transaction, is left uncommitted or un-aborted. If any, the commitment of the transaction is suspended until recoverability is achieved. When all transactions ordered prior to it are committed or aborted, the *TMO* notifies all objects in *Transaction Set* to commit the state of the transaction.

4 Preserving Atomicity Using TMOs

In this section, we show non-locking pessimistic and optimistic algorithms using *TMO*s that preserve serializability of transactions. We assume that there is neither concurrency within an object nor within a transaction as described in section 3.1. We assume that no failures occur in the system; the recovery mechanism only needs to wipe out the effects of aborted transactions. This restriction will be removed in an algorithm we are now developing, in which objects and transactions survive against site failures.

The algorithm is based on serialization graph testing (SGT) [2]; order relations between transactions generated on each object are collected by the *TMO* and consistency is preserved by negotiation between *TMOs*.

4.1 Constructing the Transaction Set

First we describe in detail the method for constructing a *Transaction Set*. This method is used with both the pessimistic and optimistic schemes for detecting and resolving inconsistency shown in next section. A *TMO* is generated when a transaction is started, and its identifier is passed by messages to all the objects accessed in the transaction. A list of objects which are accessed in the transaction, called *access history*, is piggy-backed on every message sent between objects.

All objects have to follow the following two rules to construct a *Transaction Set* and *Conflict Set*:

- The object which initiated the transaction passes the *access history* to its *TMO*, when it receives the reply from the last [3] method call of the transaction.

- An object which detects a conflict passes the *access history*, after appending its identifier to it, to all *TMO*s involved in the conflict.

All *TMO*s must follow the following two rules:

- Append all elements of the *access history* passed from the initiator of the transaction to the list of objects belonging to its *Transaction Set*.

- When an object detects a conflict, the *TMO* appends all the elements of the *access history* to the list of objects belonging to its *Transaction Set* and append the object to the list of objects belonging to its *Conflict Set*.

4.2 Detecting and Resolving Inconsistency

The order relation between transactions is generated only when two methods executed by different transactions conflict in an object. To preserve the serializability of transactions, it is important that ordering among conflicting transactions are consistent on all objects in the *Conflict Set*. We present two algorithms to preserve consistency: a pessimistic and an optimistic one.

Pessimistic Approach

In the pessimistic algorithm, when a conflict occurs, consistency orders which have been found previously in the transaction, have to be preserved. When the execution of some method in transaction A with *TMO* A, conflicts in some object, the object and the *TMO* execute the following steps.

1. The conflicting object sends a *conflict message* to *TMO* A, and *follow messages* to all the *TMO*s whose transactions accessed the object. It waits instructions from all the *TMO*s that were sent *conflict* and *follow messages*. *Follow messages* are required to detect a loop in the order between two transactions. *Conflict messages* contain the list of conflicting transactions and *follow messages* contain the identifier of A.

[3] To reduce the size of piggy-backed messages, *access history* can be passed to the *TMO* when initiator receive the reply from each method call.

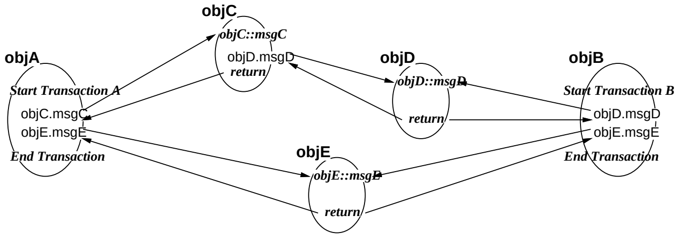

Figure 4: An Example of Transaction Processing.

2. *TMO*s which receive the *conflict message* or *follow message* check for a loop in the order of transactions. Each *TMO* has lists of *TMO*s of preceding transactions and following transactions. The loop is detected by comparing these lists.

3. When a loop is detected by a *TMO*, it negotiates with the other *TMO* involved in the loop. There are some rules to decide victims, such as a trivial one, one based on lifetime, or one based on the number of the following transactions. In the trivial rule, the transaction that executes the conflict operation first survives. In the rule based on lifetime, the transaction whose lifetime is longer survives. And in the rule based on the number of following transactions, the transaction which has the largest number of depending transactions survives.

4. When *A* is selected as a victim, it notifies all objects in *Transaction Set* to abort. If *A* is not selected, it notifies the object that detect the conflict to continue the execution.

5. When the initiator of the transaction notifies *TMO A* that the transaction has finished, *TMO A* commits the transaction using two phase commit if no preceding transaction is left uncommitted or un-aborted. If some transactions are left, *TMO A* sends *check_loop message* to all *TMO*s that manage preceding transactions. If this message comes back, the loop is constructed by more than three transactions. Therefore, *TMO*s involved in the loop negotiate to decide victims. When commit has completed, *TMO A* sends *committed message* to all *TMO*s in the *follow set* [4] to notify the commitment. The receiver of the *committed message* removes the sender from the *precedes set*[4], and commits itself if no preceding transaction is left.

In the pessimistic scheme, consistency among conflicting transactions are preserved when a conflict is detected. Therefore, if no uncommitted or un-aborted transactions are left, the transaction can commit.

Optimistic Approach

In the optimistic algorithm, consistency is not checked when a conflict is detected. The object notifies the *TMO* of the occurrence of a conflict and waits for the receipt, but not for the checking of consistency. Because the consistency is not checked at the occurrence of conflict, each transaction must check the consistency at the commitment time.

4.3 An Example

We exemplify the algorithm for preserving consistency using *TMO*. Let us consider an instance of transaction processing shown in Figure 4.

In this example, two transactions send messages to objD and objE, where conflict may occur. We consider the following case (shown in Figure 4).

1. objD receives a message from transaction A.
2. objD receives a message from transaction B.
3. objE receives a message from transaction B.
4. objE receives a message from transaction A.

[4] see section 4.4

In this case, if the two messages received at objD and objE conflict, these transactions may cause inconsistency. Our scheme can detect and solve the inconsistency by the following mechanism:

1. First, *TMO* A is generated corresponding to "start transaction A".

2. Messages are sent from objA to objC, from objC to objD, and the results are returned. The identifiers of *TMO* A, objC, and objD are piggy-backed on each message as shown in the boxes of Figure 5(a).

3. When the execution returns to the initiator of the transaction (objA), the *access history* (objC, objD) is passed to *TMO A* for constructing the *Transaction Set* (Figure 5(a)).

4. *TMO* B is generated corresponding to "start transaction B".

5. When objB sends a message to objD, the operation invoked by transaction B conflicts with the preceding operation invoked by transaction A. The conflicting information is passed to *TMO* A using *follow message* and to *TMO* B using *conflict message* (Figure 5(b)). *TMO A* appends *TMO B* to its following transactions' list, and *TMO B* appends *TMO A* to its preceding transactions' list.

   In the pessimistic scheme, objD waits the reply from *TMO*s. In the optimistic scheme, objD does not wait and the execution of transaction B is continued.

6. objB sends a message to objE, but the invoked operation does not conflict with other operations on it. The execution of the transaction is not blocked, and the result is returned to objB (Figure 5(b)).

7. When objA sends a message to objE, the operation conflicts with previous operation executed in transaction *B*. Therefore, the conflict information is passed to *TMO A* by sending a *conflict message* and to *TMO B* by sending a *follow message* (Figure 5(c)). *TMO A* appends *TMO B* to its *precedes set*, and *TMO B* appends *TMO A* to its *follow set*.

8. At least, one of the *TMO*s detects loop of the dependencies between two transactions using its local information. Then, the *TMO*s negotiate to decide the victim. The participants of the negotiation which involved in the loop are decided only from the local information. Details will be described in next section. In this case, using the trivial rule, *TMO B* is selected as a victim because transaction *B* invoked some operations on other objects later than transaction *A*.

9. The effect of transaction *B* is wiped out in each object that has received the message in transaction *B* [5]. Then, the recovery is completed and ordering information about transaction *B* is erased in *TMO A*. In this way, the inconsistency is resolved.

10. When the transaction *A* reaches to end_transaction, objA notifies *TMO A* that the transaction *A* can be committed.

11. If no transaction left uncommitted or un-aborted which should be committed or aborted prior to *TMO A*, *TMO A* instruct to all objects in *Transaction Set* to commit the update of the transaction A. The updates in transaction *A* is stored to persistent states using Two Phase Commit Protocol.

4.4 Managing Dependency and Commitment

Our scheme is based on Serialization Graph Testing. Dependency relations between transactions is considered as edges on a directed graph. However, in our scheme, we do not maintain all the information about the dependency relation. In order to detect an inconsistency, a *TMO* needs to know preceding and following transactions. Each *TMO* has the information of sets of *TMO*s: *precede set* and *follow set*. If there are some *TMO*s belonging to both sets, the inconsistent state is detected. Therefore the cost for graph maintenance is not so large.

Each *TMO* maintains the *precede set* and *follow set* as follows. When receiving a *conflict message*, the list of *TMO*s contained in the message are appended to the *precede set*. When receiving *follow message*, the sender of the message is appended to the *follow set*.

At the time of the commitment, these sets are also used. Each transaction reaching an end_transaction is committed. The status of the object is stored as persistent state. Before commit, each *TMO* needs to check that no prior transactions left uncommitted or un-aborted. Transactions which have the possibility of abort must not be committed. When no *TMO* is registered to *precede set*, the transaction managed by

[5] This cost can be reduced, using commutativity of the methods.

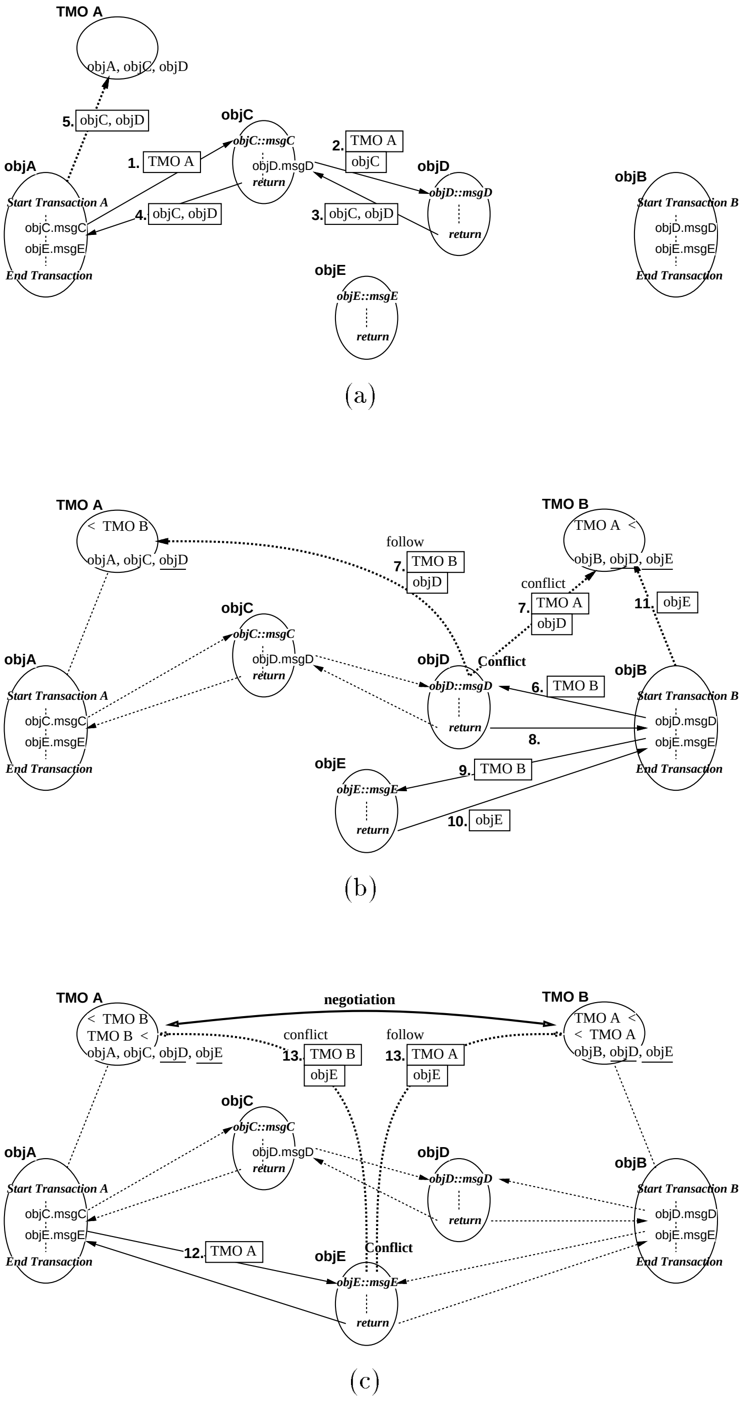

Figure 5: Process for Preserving Consistency.

the *TMO* can be committed. If some *TMO*s are registered to *precede set*, the transaction must wait for their commitment. When a *TMO* commits the transaction, the states are stored to the persistent storage, and the *TMO* sends *committed messages* to all *TMO*s belonging to its *follow set*. Every *TMO* that receives the message removes the *TMO* of the committed transaction from its *precede set*, and check the *precede set* whether there are some *TMO*s belonging to the set. If not, the same commitment operation is done.

5 Implementation

In this section, we describe the implementation of *TMO*s. In order to implement the transaction management mechanism using *TMO*s there are two possible approaches. One is to implement it under existing systems like UNIX or Mach, and the other is under object-oriented systems such as Apertos [19, 18]. In the former case, we can implement our system as a virtual machine (e.g. Smalltalk-80), or with objects as processes and *TMO*s as servers. In the latter case, we can implement *TMO*s as meta-level objects.

We have implemented our prototype system under UNIX. Objects are implemented as processes, and *TMO*s are implemented as entries in a server. We have confirmed the correctness of the algorithm and measured the costs of transaction management such as creation of a *TMO*, negotiation to decide victims, commitment, abort, and retry.

The current prototype system runs in the distributed environment with SPARC Stations/Sun OS 4.1.1 using TCP/IP. The basic construction of the system is shown in Figure 6. *TMO* Server is build

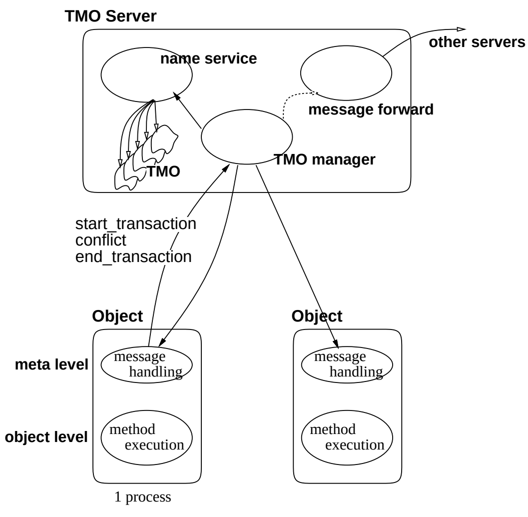

Figure 6: The Structure of *TMO* Server

with three modules. The main module is the *TMO* Manager which manages *TMO*s and actually doing negotiations between *TMO*s; the name server manages the mapping of object identifiers and object names, *TMO*'s id and location in the server; the message forwarder is used to forward messages to remote *TMO* server. At present, our implementation is not provided with remote *TMO* servers. Thus, our system implements as communication and method control routines in each object("message handling" in the figure 6) and *TMO* servers.

We have measured the costs of *TMO* creation and negotiation, which are shown in Table 1. Currently the *TMO* server is implemented as a single server. The cost of *TMO* creation is the time between sending start_transaction to the *TMO* server and the receiving of the reply from the server at the object. There are two columns in the table for *TMO* creation, one for when the *TMO* server is located in the local node, and the other in a remote node.

The cost of negotiation is the time that needs to detect and resolve inconsistency. The times shown in the table are the average time between detecting conflict on an object and receiving the decision after negotiation between two *TMO*s. We use two negotiation rules: the trivial rule and the one based on lifetime. In the trivial rule, when a loop in the order is detected, the transaction that executed the conflict operation first survives. In the lifetime based rule, the transaction whose lifetime is longer survives. For sake of the comparison, the time need for null call on TCP/IP is shown in the table.

Table 1: Cost to Guarantee Consistency

null call(remote)	1.36ms
TMO creation(local)	2.93ms
TMO creation(remote)	3.18ms
negotiation(trivial)	2.62ms
negotiation(life time)	2.66ms

We think the costs are within an acceptable order. We expect that our scheme will exhibit improved performance by one order of magnitude on the Apertos object-oriented operating system.

6 Discussion

In this section, we compare our scheme to previous works.

6.1 Comparison with Local Atomicity and Cooperative Atomicity

Features and problems of local atomicity and cooperative atomicity were discussed in subsection 2.1. In comparison with the local atomicity scheme, our scheme ensures the serial order of operations to be consistent for the smallest set of objects which actually conflict. Our scheme does not use global information as opposed to local atomicity. Moreover, our scheme has the advantage that it does not need the order relation on all objects which were previously accessed but only the information of objects on which conflicts occur. So, the cost for collecting order relations is reduced.

In cooperative atomicity, objects never propagate conflict information except when the conflict is detected, because objects remain passive after executing a method. Hence, in order to check consistency, an object which detects a conflict must collect information from all objects that were accessed before in that transaction. On the other hand, in our scheme, conflict relations generated later are notified to all the *TMO*s of conflicting transactions. Therefore, a *TMO* knows the objects with which a conflict occurred, and thus inconsistent orders can be detected by a *TMO* by checking the orders on the objects in the *Conflict Set*. Furthermore, when comparing the optimistic schemes, inconsistency is detected only at the commitment time in cooperative atomicity. However, *TMO*s detect inconsistency concurrently with the execution of transactions, so the period for which the system remains in an inconsistent state is reduced.

6.1.1 Selection of Victims

In the scheme based on cooperative atomicity, a victim which resolves inconsistency is statically determined and may not always be the most suitable one. In practical situations, it is useful that the most suitable victim can be chosen dynamically. For example, a long transaction is expensive to abort. If there are multiple loops in orders between transactions, the transaction that causes the multiple loop should be selected as the victim. In our scheme, *TMOs* can negotiate to choose an appropriate victim. In the other words, a *TMO* can have information on transactions, and thus collect the cost information for abort in *Transaction Set*. Therefore, the characteristics of transactions and characteristics of objects in transactions can be reflected to the control of the transactions.

In the scheme based on cooperative atomicity, since any object can become a coordinator to detect inconsistency, each object must have all the functions for preserving consistency. In our scheme, only *TMO*s have to provide the ability to preserve consistency and to determine which transaction should be aborted. The policy of the transaction management is clearly separated from its mechanism within *TMO*s. Thus, we achieve high modularity.

6.1.2 Global Loop of Dependencies

In our scheme, a loop constructed by more than three transactions (this is called here a global loop) is detected at commitment time. In cooperative atomicity, in order to detect a global loop, atomic broadcast is used recursively until information from all objects related to the transactions which compose the loop is received. Using *TMO*s , in order to detect a global loop, a *TMO* sends *check_loop messages* to other *TMO*s in the *precedes set* at commitment time, and the receivers repeat it until reaching the earliest scheduled transactions or detecting a loop. If a message comes back to the sender, a global loop is detected. This scheme does not require broadcast, and fewer messages are necessary to detect the loop.

6.2 Response Time of Transactions

We now compare our scheme with cooperative atomicity from the viewpoint of blocking time when a conflict occurs, and the number of messages for preserving the consistency in the pessimistic algorithm. Let the communication time between an object which discovers a conflict and the objects in the *Transaction Set* be $\{p_0, \cdots, p_n\}$. Let the communication time between an object which discovers a conflict and *TMO*s which manage conflicting transaction be $\{t_0, \cdots, t_m\}$. In the cooperative atomicity scheme, the execution of the transaction is blocked for $4 \times max(p_0, \cdots, p_n)$, and each object in *Transaction Set* is blocked for $2 \times max(p_0, \cdots, p_n)$, when conflict is detected. In our scheme, since the object which detects a conflict sends a *conflict message* and *follow messages* to *TMO*s and waits for the instruction from all *TMO*s, the object is blocked for $2 \times max(t_0, \cdots, t_n)$ for checking consistency, and other objects in *Transaction Set* are not blocked. Using the trivial rule, the cost of deciding victim is included in this estimative. To realize the same criterion to the scheme based on cooperative atomicity, negotiation is unnecessary. Therefore, response time is improved. However, if we pay the cost for negotiation , we can select suitable victims.

Let the number of objects in *Transaction Set* be m, and let the number of transactions which is conflicting with a transaction be n. The amount of messages for detecting inconsistency is $4 \times m$ in cooperative atomicity scheme. In our scheme, the amount of messages for detecting inconsistency is $2 \times (n+1)$, since the object sends a *conflict message* and n *follow messages* and receive reply from the *TMO* which manages the transaction. When inconsistency is detected, some messages are exchanged for negotiation. In the trivial case, a victim is selected based on the order of transactions with less cascading aborts. In this case no more messages are exchanged, since the *TMO* has all information to decide a victim.

Finally we recapitulate some of the advantage of our scheme. The response time can be decreased, since our scheme does not use locking. Inconsistency can be detected at an earlier time by *TMO* even when using the optimistic scheme, since *TMO*s can run concurrently with ordinary objects. Flexibility is increased, since the characteristic of transactions can be reflected to its management. Moreover, since transactions are dynamically managed in finer granularity by cooperation among *TMO*s, we believe our scheme is suitable for the open-ended, large-scale distributed system.

6.3 Comparison with ERMS

ETM (ERMS Transaction Manager) provided in ERMS (Eden Resource Management System) is a transaction manager which manages one transaction. ETM is similar to *TMO*; like an ETM, one *TMO* is created per transaction. However, an ETM contains lock and version information of the transaction, and transaction operations such as lock, versioning, commit, abort, recovery are processed by the system level Lock Manager and Data Manager. In our scheme, a *TMO* contains all the dependency information of the objects in the transaction, and can decide progress or delay execution and commit or abort. Data management and recovery facilities are supported by each object.

*TMO*s exchange their dependency relations, and negotiate with each other to determine the transaction to abort in an inconsistent case. *TMO*s use the information of transactions such as lifetime, priority, and retry times, to decide victim, and try to maintain system consistency by itself. ETMs do nothing by themselves. This means that using *TMO*, neither System Transaction Manager nor System Data Manager are necessary, and consistency is managed by making groups of objects that relate to the transaction dynamically. Comparison of *TMO* and ETM is summarized in Table 2.

7 Conclusion

In this paper, we proposed a scheme for guaranteeing the consistency of objects in very large distributed environments. The scheme is based on Serialization Graph Testing. We introduced a *TMO* (*Transaction Management Object*) which is created for each transaction. The consistency of transactions is managed by negotiation among *TMO*s. Since we employ a non-locking algorithm and assume no global entity, our scheme is suitable for very large distributed systems. Furthermore, we can reflect the characteristics of transactions and objects to transaction management in negotiation among *TMO*s. Thus, we obtain both efficiency and flexibility from our scheme. We assumed in this paper that concurrency within objects and within transactions does not exist; these restrictions are now being removed and the result will be presented in a separate paper.

We implemented a prototype system as a single server under UNIX. From our preliminary measurement of various system costs, our scheme was proved to be efficient. Furthermore our scheme increased flexibility and survivability of transactions and improved response time.

*TMO*s can naturally be modelled as meta-level objects. Thus, we are interested in implementing *TMO*s as meta-level objects in an object-oriented distributed operating system. Particularly, since the Apertos operating system provides reflective capability [19], we are much interested in implementing our system under the Apertos operating system.

Acknowledgments

We would like to thank Hideaki Okamura, Hiroko Fujii, Atsushi Shionozaki and Vasco Vasconcelos for their valuable comments.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System : A Technical Review. *IEEE Transactions Software Engineering*, SE-11(1):43–59, 1985.

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

[3] Maurice P. Herlihy and Jeannette M. Wing. Avalon : Language Support for Reliable Dis-

Table 2: Comparison between *ETM* and *TMO*

	ETM	*TMO*
functions	both concurrency control and failure recovery	policy of concurrency control and decision of the victims
entity	lock table and table of committed version	object which has information of objects related to the transaction and policy of transaction management
concurrency control	Two Phase Locking	SGT, Optimistic & Pessimistic
characteristics	consistency is guaranteed by a lower layer function	exchange information and negotiate among *TMO*s to guarantee consistency

tributed Systems. In *17th Symp. on Fault-Tolerant Computing*, pages 89–94, 1987.

[4] Maurice P. Herlihy and Jeannette M. Wing. Axioms for Concurrent Objects. In *Proceeding of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, 1987.

[5] B. H. Liskov. On Linguistic Support for Distributed Programs. *IEEE Transactions Software Engineering*, SE-8(3):203–210, May 1982.

[6] B. H. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the* 11^{th} *Symposium on Operating System Principles*, pages 111–122. ACM, Nov. 1987.

[7] C. R. Manning. Introduction to Programming Actors in Acore. In *Actor Systems*. MIT Press, 1990.

[8] J.E.B Moss. *Nested Transaction: An approach to reliable distributed computing*. MIT Press, 1985.

[9] Tatsuo Nakajima. *Atomicity in Multiversion Atomic Object*. PhD thesis, Keio University, Jan. 1990.

[10] Graham D. Parrington and Santosh K. Shrivastava. Implementing Concurrecy Control in Reliable Distributed Object-Oriented Systems. In *ECOOP '88*, 1988.

[11] Calton Pu and Jerre D. Noe. Design and Implementation of Nested Transaction in Eden. In 6^{th} *Symposium on Reliability in Distributed Software and Database Systems*, pages 126–136, 1987.

[12] Santosh K. Shrivastava, Graemme N. Dixson, and Graham D. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, 1991.

[13] Alfred Z. Spector. Distributed Transaction Processing and The Camelot System. Technical Report CMU-CS-87-100, Carnegie-Mellon University, Jan. 1987.

[14] W. E. Weihl. *Specification and implementation of atomic data types*. PhD thesis, Massachusetts Institute of Technology, 1984.

[15] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transaction on Computer*, 37(12):1488–1505, Dec. 1988.

[16] Yasuhiko Yokote. *A Study on Object-Oriented Concurrent Programming Languages*. PhD thesis, Keio University, 1988.

[17] Yasuhiko Yokote. The Muse Reflective Operating System: The Concept and Its Implementation. In *To be appeared in OOPSLA '92 Proceedings*, 1992.

[18] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. Technical Report SCSL-TR-91-002, Sony Computer Science Laboratory, Feb. 1991. also appeared in Operating Systems Review, Vol.25, No.2, April, 1991.

[19] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Third European Conference on Object-Oriented Programming*, pages 89–106, 1989.