18

Nested Transaction and Concurrency Control Services on CORBA

Kai-Chih Liang[*], *Shyan-Ming Yuan*[*], *Deron Liang*[**], *Winston Lo*[***]

Abstract

This paper presents the exploration and integration of the object transaction service and concurrency control service based on CORBA. We provide not only the flat transaction model but also the nested transaction model. Nested transaction provides isolation of failures and enhances concurrency of long-lived transaction. Issues on supporting nested transactions in both the object transaction service and the concurrency control service are discussed. We also reveal the necessary overhead introduced by these two services in our implementation.

Keywords

Object Transaction Service, Concurrency Control Service, CORBA, Distributed Object Computing.

1 INTRODUCTION

Transaction paradigm is useful to build robust systems. As we can see, transaction processing plays a very important role in the business. Data sharing is common in the distributed environment. Concurrent accesses to the same data must be properly controlled to keep data in consistent states. These two important concepts have been included in the OMG's (Object Management Group) common object services. They are Object Transaction Service (OTS) and Concurrency Control Service (CCS).

[*] Department of Computer and Information Science National, Chiao-Tung University, Hsinchu, Taiwan, R.O.C.
[**] Institute of Information Science Academia Sinica, Taipei, Taiwan, R.O.C.
[***] Department of Computer Science and Information Engineering, Feng Chia University, TaiChung, Taiwan, R.O.C.

J. Rolia et al. (eds.), *Open Distributed Processing and Distributed Platforms*

The OMG's CORBA (Common Object Request Broker Architecture) is the industrial standard of distributed object technology. (OMG 1992, OMG 1995a) The core of CORBA is the object request broker (ORB) that serves as an interconnection bus between distributed objects. OMG defines a set of object services upon the ORB. These services are specified in the common object service specification (COSS). (OMG 1995b)

After the OTS standard being adopted by OMG in the middle of 1995, traditional TP monitors follow this standard to migrate into OOTP monitors. (Hitachi 1996) Meanwhile, ORB vendors want to provide the object transaction services with their products. But, few of them claim that their products support nested transactions. In our first system, (Yue-Shan 1996) it supports the flat transaction mode OTS over the ORB. This is the pioneer toward the distributed object transaction world. From the experience of developing our previous system, we are encouraged to extend it to provide both the flat and the nested transaction models. In addition, we design and implement another service, the CCS, to facilitate transactional application development. Our system is developed upon one of the well known ORB products, the IONA's Orbix for Windows NT.

The goal of this research is to explore and implement the object transaction service and concurrency control service so that application object programmers can reuse these services to build reliable and robust distributed software efficiently. The OTS supports not only the flat transaction model but also the nested transaction model. The CCS provides multiple granularity, flexible lock duration locking service over a distributed environment. These two services together guarantee the essential transaction properties. (Jim 1992)

The system diagram of our system is shown in the Figure 1. A client first begins a transaction by issuing a request to an object in OTS, which constructs a corresponding transaction context. Then, it performs transactional operations by issuing requests to resource objects. Resource objects that in turns shall register themselves to the OTS and acquire locks from the CCS. Eventually, the client ends the transaction by issuing another request to the OTS to trigger the atomic commit procedure that coordinates all objects involved in this transaction on behalf of the client.

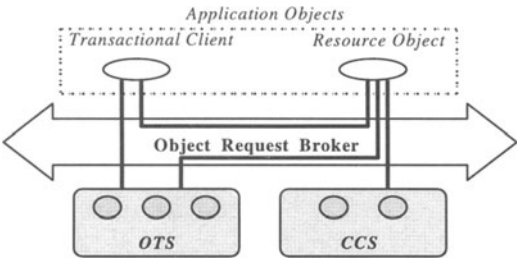

Figure 1. System Diagram

Compares to other commercial products, our system emphasizes on the nested transaction support. Supporting nested transaction is a big quest. Both the OTS and the CCS have to take care on something that does not appear in the flat transaction. Besides the implementation issues, we have evaluated the extra overhead that must be taken when employing the OTS and the CCS.

2 SYSTEM OVERVIEW

In this section, we give an overview of concepts of the object transaction service and the concurrency control service. Features and functionalities of each service will be discussed.

2.1 Object Transaction Service Overview

The Object Transaction Service brings the transaction paradigm, which is essential to developing reliable distributed applications, and the object paradigm, which is key to productivity and quality in application development. It provides transaction synchronization across the elements of a distributed client-server application. (Iain 1995)

The OTS supports two distributed transaction models: the flat transaction model and the nested transaction model. A flat transaction is a transaction that cannot have a child transaction. Nested transactions, on the other hand, allow subtransactions embedded in the current transaction. The typical scenario of the nested transaction in OTS is shown in the Figure 2. Note that we employ the explicit mode of the OTS. (OMG 1995b)

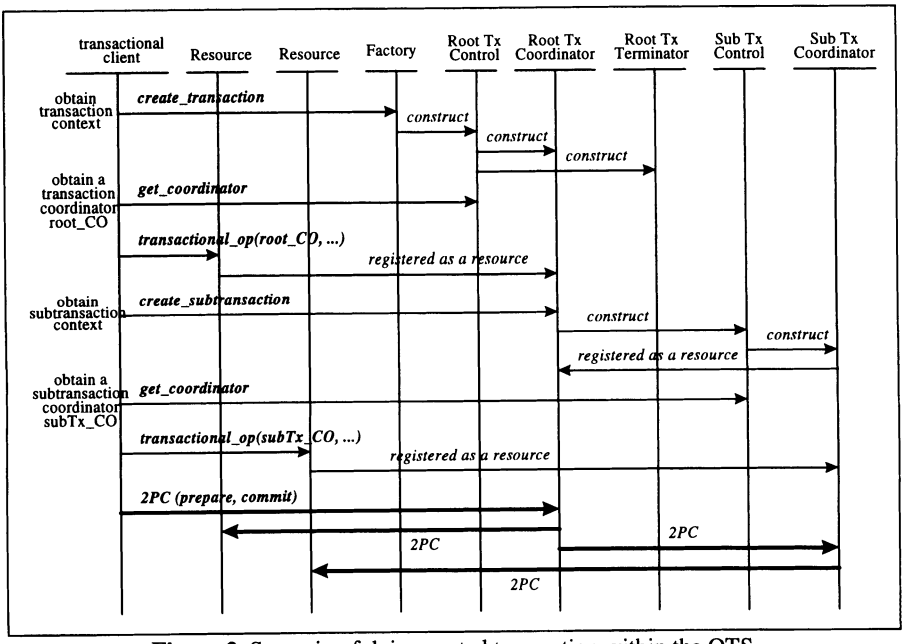

Figure 2. Scenario of doing nested transaction within the OTS.

Nested transactions provide a finer granularity of recovery than flat transactions. That is, when one of the subtransactions of a parent transaction fails, only the subtransaction is rolled back. The parent transaction has the opportunity to correct or compensate for the failure and finish its work. In addition, subtransactions can be executed in parallel without the risk of inconsistent results.

A subtransaction is similar to a top-level transaction that the changes made on behalf of a subtransaction are either committed in their entirety or rolled back. However, when a subtransaction is committed, the changes remain contingent upon commitment of all its ancestors. A transaction cannot commit unless all of its children are completed. When a transaction is rolled back, all of its children are rolled back.

2.2 Concurrency Control Service Overview

The purpose of a concurrency control service is to mediate concurrent accesses to an object such that the object is in a consistent state. In the distributed environment, there must be some places that keep the access information in order to govern concurrent accesses. OMG defines the concurrency control service as the standard object service. It is a lock-based concurrency control service.

The CCS does not define what a resource is; It is up to clients to define resources and to manage the mapping between resources and locks. Typically, an object can be regarded as a resource by the client. This increases the flexibility of lock granularity.

The level of isolation can be changed by adjusting the lock duration of the resource held by the transaction. Typically, a transaction will retain all of its locks until the transaction is completed. The CCS allows clients to make decision about their own lock duration. In addition, it provides a Lock Coordinator object that drops locks of the same transaction together.

The CCS defines five kinds of lock modes: *read, write, upgrade, intention read,* and *intention write*. Read and write lock modes provide the conventional multiple-readers-single-writer policy. An upgrade-mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock that occurs when two or more clients attempt to read and then upgrade the same resource. Intention locks are locks that allow variable granularity of locking. It is useful to exploit the natural hierarchical relationship between locks. Using variable granularity locking, a client first obtains intention locks on the ancestor of the required resource, and then acquires more strict lock, say read or write locks, on the child of the resource. The lock modes compatibility is summarized in the Table 1.

Table 1. CCS Lock Compatibility

Granted Mode	Requested Mode				
	IR	*R*	*U*	*IW*	*W*
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

(is used to indicate conflicts)*

The CCS has extensive lock modes that allow more concurrency levels than traditional ones. In addition, it adds supports on nested transactions to further extends the concurrency level. If the nested transaction is used, lock conflicts within a transaction family are treated somewhat differently from those between unrelated transactions. A parent transaction cannot

abort without causing all of its children to abort, while a child transaction that ends successfully cannot abort without causing its parent abort.

3 SYSTEM ARCHITECTURE

The objective of this research is to implement and integrate the object transaction service and the concurrency control service defined in the COSS. (OMG 1995b) We'll illustrates the design of our system from the highest level of abstraction to the detailed system architecture.

3.1 Overview

Our system is based on the OTS and the CCS specification. The development environment is Windows NT 3.51 with Visual C++ 2.0 compiler. The underlying ORB connection is Iona's Orbix 2.0. (IONA 1995a, IONA 1995b) We implement these two services on CORBA environment to facilitate easy development of distributed transactional applications. Our current implementation supports both the flat and nested transaction model.

Figure 3. illustrates the four major parts of our system. The transaction originator is intended to use the recoverable server in transactional behavior. It is often referred to as a transactional client. The recoverable server provides recoverable objects that can be used by transactional clients. Our system provides OTS and CCS for application programmers to facilitate their easy use of transaction paradigm.

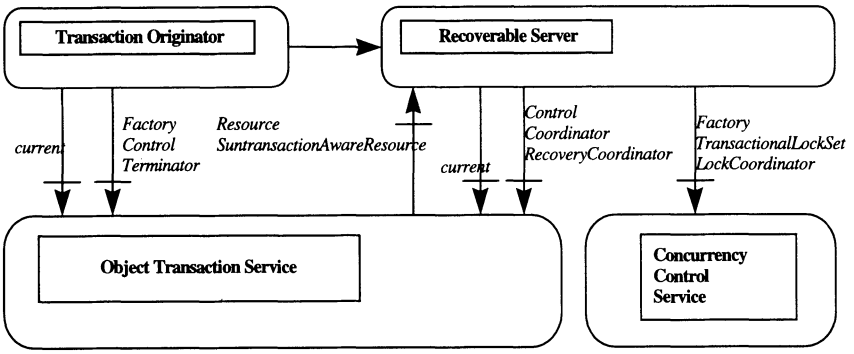

Figure 3. Top-level Functional Diagram

We implement the OTS and the CCS as the OTS Manager and the CCS Manager. Figure 4. shows the components and relationships of the OTS and the CCS. Note that we introduce an additional component, *LockCoordinatorManager*, in the CCS Manager. This component is private to the CCS Manager itself. We introduce this new component to solve the Transaction-LockSet-LockCoordinator mapping problem gracefully. The detail will be examined later.

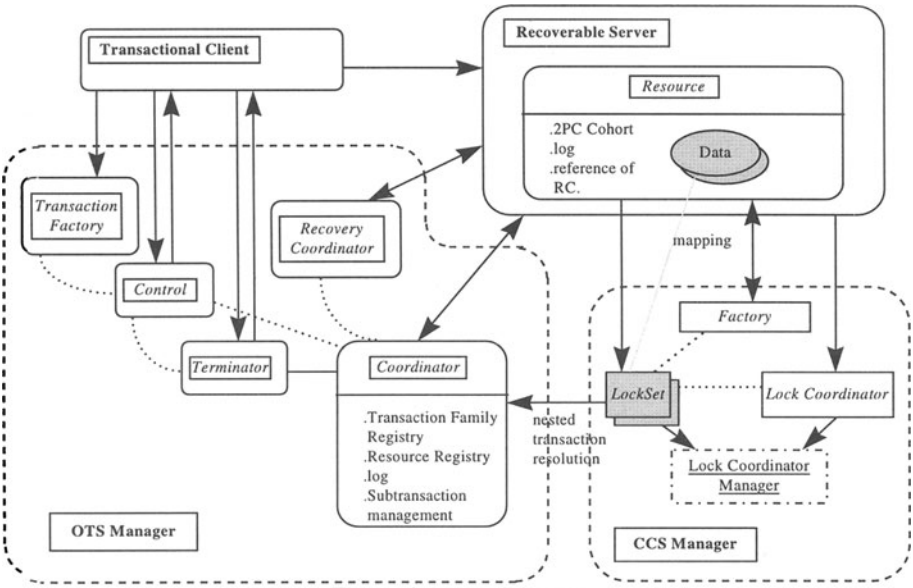

Figure 4. System Diagram

3.2 Design of the Object Transaction Service

Our OTS extends our previous system in the aspect of nested transaction. (Yue-Shan 1996) To create a subtransaction, a transactional client should first get the Coordinator object. Then, it issues the create_subtransaction request to the Coordinator. The Coordinator creates a Control object for the representation of transaction context of the subtransaction. This Control object creates a Terminator object and a Coordinator object as the Control object of the parent transaction does. The Coordinator of the subtransaction should register itself to the parent coordinator as a resource object. Finally, the parent transaction's Coordinator returns a reference of the Control object of the subtransaction to the transactional client.

As to the nested transaction termination, the coordinator of the subtransaction is also a participant of the parent transaction. It launches another 2PC process on its subtransaction participants and reports a joint decision to the parent transaction.

3.3 Design of the Concurrency Control Service

Recoverable objects may isolate one transaction from another by using the CCS. The recoverable object acquires locksets with desired lock modes from the CCS. If the desired lock is not granted, the recoverable object blocks the client's request. In order to enforce the two phase locking (2PL) policy, the CCS provides a LockCoordinator object that drops all locks on behalf of a transaction at a time. For the purpose of relating the LockCoordinator object with the transaction's lockset, we introduce a LockCoordinatorManager. Figure 5. depicts the typical process of using the CCS.

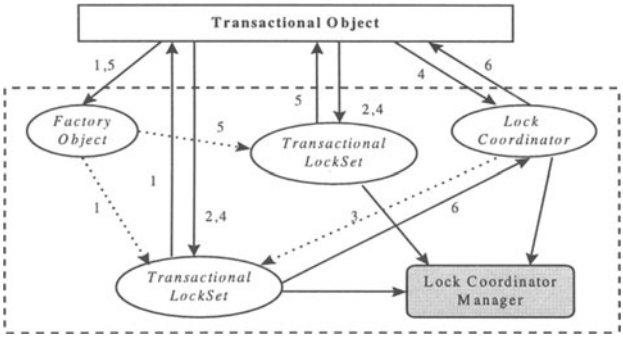

Figure 5. Design of the Concurrency Control Service

When a recoverable object wants to acquire locks on behalf of its resources, it needs to create locksets first. The recoverable object issues the create_lockset request to the Factory object. The Factory object, then, creates a TransactionalLockSet object and returns the reference of this object to the recoverable object. The TransactionalLockSet object will register itself to the LockCoordinatorManager.

Related locksets are useful when you want to drop their locks of the same transaction together. Creating a related lockset is similar to creating a lockset. The recoverable object issues a create_related_lockset request to the Factory object. The Factory object creates a new lockset. And this new lockset registers the related lockset information to the LockCoordinatorManager.

After acquiring a lockset, the recoverable object can perform some operations on it, such as lock, unlock, etc. If the TransactionalLockSet object finds that the incoming request is associated with a new transaction context, it will create a LockCoordinator object for it and register this information on the LockCoordinatorManager.

In the typical scenario of two phase locking, especially the strict two phase locking, the transaction will drop all locks after the 2PC is done. It is the performance consideration that the shrinking phase of 2PL should be fast enough to minimize the response time of the transactional client. The LockCoordinator object of the CCS serves for this purpose. The recoverable object should get the LockCoordinator object first by issuing a get_coordinator request to the TransactionalLockSet object. Then, by using the LockCoordinator object, the recoverable object issues a drop_locks request to drop all locks with respect to the transaction.

4 IMPLEMENTATION ISSUES

This section describes some significant implementation issues of the OTS and the CCS.

4.1 Nested Transaction Support

According to the OTS specification, the nested transaction may be used in the following scenario:

1. The transactional object first creates a new transaction nested to the current one by issuing the Coordinator::create_subtransaction() method call.
2. The coordinator of the newly created subtransaction must register itself as a resource object to the parent transaction coordinator.
3. If the transactional object wants to terminate the transaction, it may issue a commit request to trigger the two phase commit protocol.
4. The parent transaction's coordinator then coordinates the commit process by sending messages to all its registered resources as well as the subtransaction coordinator.

But, here comes a problem. According to the specification, the coordinator interface does not inherit from the resource interface. Thus, according to the concept of object-oriented technology, the subtransaction coordinator cannot pass itself as a resource object when registering the subtransaction. To overcome this problem, we implement a special resource object and add some internal interfaces (interfaces that are private to the object server, not to the client). This special resource object will be created when a subtransaction is created. It acts as a proxy of the subtransaction coordinator to the parent transaction coordinator. When it receives the 2PC requests from the parent transaction, it forwards (or delegates, in object-oriented terminology) the requests to the subtransaction coordinator by calling the internal interfaces. As the subtransaction coordinator returns the vote, this special resource replies the vote to the parent transaction coordinator. Thus, we solve this quiz without having to modify interfaces described in IDL. Note that modifying the interfaces described in IDL usually means modifying the external behavior of the service, and it will ripple every objects. Figure 6. depicts this scenario.

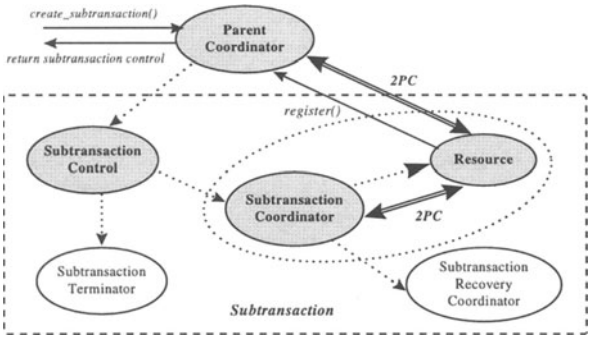

Figure 6. Implementation of nested transaction support of the OTS

4.2 Multithread Support

According to the CCS specification, a lock request would be blocked if it conflicts with the existing locks. This implies that our CCS Manager must be presented as a concurrent server. From the help of the Orbix for Windows NT, we design and implement our CCS Manager as a multithread concurrent server. We create a new thread for each request. Thus, a blocked thread will not block others.

4.3 Lock Coordinator Manager

As mentioned in the system overview, there is a critical action in the CCS: the Transaction-LockSet-LockCoordinator mapping. Given a transaction and a chain of related locksets, we want to find a LockCoordinator object. This LockCoordinator object can drop all locks in the chain of related locksets on behalf of the given transaction. Figure 7. depicts this mapping.

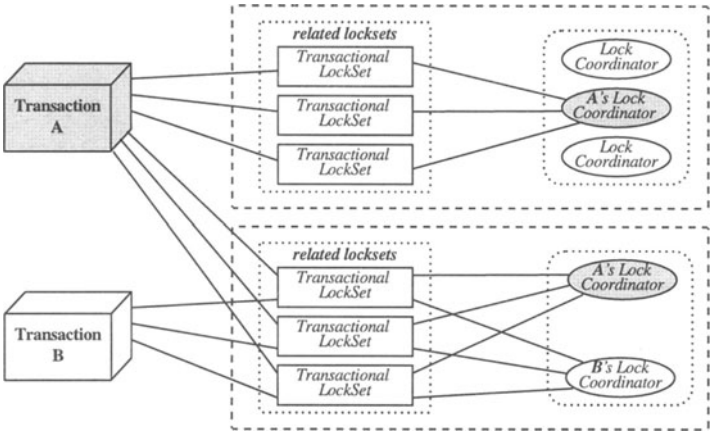

Figure 7. Relationships among Transaction, related LockSet chain and LockCoordinator.

In order to find the LockCoordinator object, we have to keep some information. Instead of adding extra internal interfaces to LockSet/TransactionalLockSet object to enforce the distributed management of the mapping information, we introduce an object to centralize the mapping information. It is the LockCoordinatorManager object. TransactionalLockSet or LockSet objects use the LockCoordinatorManager object to manage their relationship information.

5 PERFORMANCE

We'll examine the necessary overhead introduced by the OTS and the CCS via evaluating the response time of each critical operation in these services. It shows that our system has acceptable performance in many aspects.

5.1 Overview

Our system provides only part of basic components for transaction processing. It has no meaning for us to take the full TP benchmark testing. Instead of showing our system performance by taking benchmarks, we show it by identifying the overhead. We employ three servers. They are OTS server, CCS server, and NullRO server. The NullRO server is a

collection of recoverable objects (RO) that do nothing but just vote Commit in the two phase commit protocol (2PC).

The test is taken under a LAN configuration. Two Pentium-based PC workstations are used : One runs the OTS and the CCS, the other runs the Null RO and testing clients. System performance is judged in terms of response time observed by client objects. This measurement includes the computing and networking overhead that will be suffered in the real situation.

To make the evaluation more reasonable, we define a stable state in which the system is measured. A state is stable if the system has approximately the same performance for the same test cases. To make sure the system has reached the stable state, we run some light-weight test cases to "warm up" the system before each measurement.

5.2 Performance of the Concurrency Control Service

We first discuss the cost of CCS operations in terms of response time. Because of the measurement precision, the cost of CCS operations can not be measured by just taking timestamps before and after a single action. Instead, we take the 1st timestamp, do the same operation several times, and take the 2nd timestamp. The result is summarized in the Table 2.

Table 2. CCS Performance Summary

LockSet Creation (*msec*)										
# lockset exist	20	40	60	80	100	120	140	160	180	200
create	0	3	3	10	6	6	10	10	16	13
create related	6	13	10	16	10	13	16	13	20	20
lock / try_lock operations (*msec*)										
# transaction exist	10	20	30	40	50	60	70	80	90	100
incompatible try_lock	52	54	55	55	55	56	56	57	58	60
lock	22	22	22	22	22	23	23	23	23	23
drop_locks (*msec*), **drop all locks on behalf of a transaction*										
# lockset	10	20	30	40	50	60	70	80	90	100
10 lock per lockset	80	160	231	311	**380**	461	541	611	681	751
# locks in a locket	10	20	30	40	50	60	70	80	90	100
total 50 locksets	380	380	380	380	380	380	380	380	380	380

We judge the cost of the lockset creation by measure its marginal cost. In other words, we measure the response time of creating one lockset when there are already some number of locksets in the system. It is reasonable that the cost of creating a related lockset is more expensive than that of creating a simple lockset. On the average, the marginal cost of creating an additional lockset is below 10 msec, and the marginal cost of creating a related lockset is below 15 msec. It is an acceptable and reasonable result.

The *try_lock* operation tells whether the request lock conflicts with existing locks. The cost is too light when the request lock is compatible with the existing locks. However, when the request lock is in the incompatible mode, we must pay more efforts on resolving the relationship between transactions. Extra message overhead is introduced, which makes our lock resolution more expensive. As to the cost of *lock* operation, it does not make sense to deal with lock operations in incompatible modes. Therefore, we only measure the cost of lock operations in compatible modes. In usual cases, this cost is nearly a constant.

Our design shows that we can drop locks of the same transaction in constant time, no matter how many locks it does. The cost of *drop_locks* is only proportional to number of locksets that a transaction acquired.

5.3 Performance of the Object Transaction Service

Now, we discuss the performance of the two phase commit (2PC) process, the most significant operation of the OTS.

We evaluate the 2PC overhead by introducing the NullRO server. A null resource object simply vote Commit when it is triggered by the 2PC process. Thus, the response time which is measured on the client object is the net overhead of the 2PC process. Figure 8. shows the average cost of "2PC"ing a RO in both flat and nested transaction model when there are some number of ROs in the transaction family. The average cost on a RO in the nested transaction is derived by averaging the measurement in different nested transaction configurations, such as the single level transaction tree and the degenerated transaction tree, of the same number of ROs. In fact, different nested transaction configuration almost does not effect the per-RO cost. It is shown that the 2PC in nested transactions has slightly higher cost than in flat transactions.

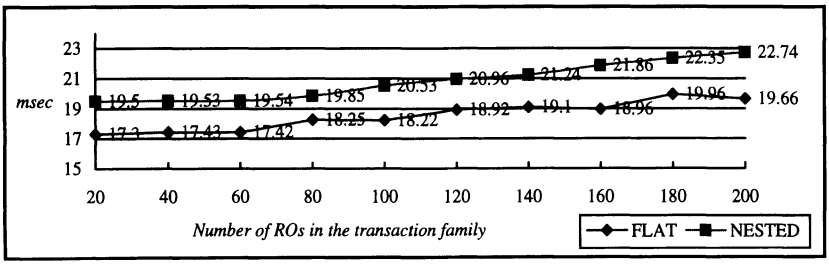

Figure 8. Average cost of a RO in the 2PC

6 CONCLUSION AND FUTURE WORKS

The concept of transactions is useful not only in database applications, but also in building robust distributed mission-critical applications. Over these years, it has been shown that distributed applications can be built effectively using distributed object technology for its strong support of heterogeneous platforms. We have implemented both the object transaction service and the concurrency control service based on the CORBA specification. Transactional applications developed with support of our OTS and CCS implementations are able to maintain the essential transaction properties of the data objects.

Our current implementation of OTS supports synchronous 2PC. That is the Coordinator is blocked and waits for the response for each RO call. There are some other ways to enhance the performance of the 2PC. Asynchronous 2PC will not be blocked after each RO call. It will do polling after sending all requests to ROs. Thus, asynchronous 2PC increases the concurrency level of the 2PC process. Multi-threaded coordinator implementation is also a way to increase the concurrency level of the 2PC process. Each thread waits for the response from single RO.

Although we have finished most OTS and CCS features, there are some ways to go further. In the object-oriented's world, the persistency property is more nature than before. We wish to introduce the object-oriented database into our system as the persistent store of objects. Integrate this system with other common object services, such as *Persistent Object Service*, *Query Service*, and *Relationship Service*, to provide a more comprehensive environment for distributed object application programmers.

Acknowledgement

This work was supported by the National Science Council of R.O.C. under Contract No. NSC 86-2213-E-035-005 at Feng Chia University.

REFERENCES

Hitachi. (1996) TP-Broker Whitepapers. http://www.zoosoft.com/jp1/wht-tpbr.html.

IONA Technologies Ltd. (1995a) Programming guide: Orbix 2 distributed object technology. Release 2.0.

IONA Technologies Ltd. (1995b) Reference Guide: Orbix 2 distributed object technology. Release 2.0.

Iain Houston. (1995) Transactions: Who needs them? http://204.146.47.71/objects/ owsf.html. IBM Corporation.

Jim Gray and Andeas Reuter. (1993) Transaction Processing Concepts and Techniques. Morgan Kaufmann Publishers, Inc.

Object Management Group, Inc. (1992) Object Management Architecture Guide.

Object Management Group, Inc. (1995a) Common Object Request Broker Architecture and Specification. Reversion 2.0.

Object Management Group, Inc. (1995b) Common Object Services Specification.

Yue-Shan Chang, Yu-Ming Kao, Shyan-Ming Yuan, and Deron Liang. (1996) An Object Transaction Service based on CORBA Architecture. Proceeding. of 1996 IFIP/IEEE International Conference on Distributed Platforms.

BIOGRAPHY

We are a special workgroup concerning with the distributed object technology. Members of our group come form NCTU, FCU, AS of R.O.C. Our current researches are ORB developement, IIOP bridge design, and implementing the common object services. In addition, we want to introduce the fault tolerant service on CORBA. For more detail information, please refer to our homepage, http://plato.cis.nctu.edu.tw/CORBA/.