An Object Transaction Service in CORBA

Deron Liang
Institute of Information Science
Academia Sinica
Taipei, Taiwan, 11529
R.O.C.

Y. M. Kao & S. M. Yuan
Department of Computer Science
National Chiao-Tung University
Hsin-Chu, Taiwan 31151
R.O.C.

Key words:
Transaction service, object-oriented programming, distributed computing environment, distributed object services, OMA, CORBA.

Abstract

The concept of transactions is not only indispensable in database applications, but also useful in building robust software for mission-critical applications. This paper presents an implementation of the Object Transaction Service (OTS) based on CORBA specification. Transactional applications developed with the support of our OTS implementation are able to assure the ACID properties even in the presence of node crashes, software system failures and process hangs. The preliminary results obtained from the experiments on Sun workstations with Orbix 1.3 show that the overhead due to the OTS service is satisfactory for most applications.

1. Introduction

The concept of *transactions* has been successfully applied in database applications to model many business events. As the client-server model been accepted as the common programming paradigm in distributed systems, transactions have been found as a useful application design philosophy to build reliable mission-critical applications[1,2]. Recently, the *distributed object technology* has been recognized as an effective distributed programming paradigm since this emerging technology is designed to support heterogeneous computing over networks with embedded object-oriented concept[3]. As a result, it is highly desirable if a distributed object programming environment with the support of transaction service is available to implement client-server based applications.

The most mature distributed programming environment designed for transactional

applications is *Transaction Monitor*. Popular products are CICS [4], Tuxedo [5], and Encina[6]. These products all support transaction service over heterogeneous platforms. They are, however, built with traditional programming paradigm instead of object-oriented programming paradigm. The emerging distributed object technologies including Microsoft's COM[7], IBM's DSOM [8, 9] and OMG's CORBA [10] are all built with object-oriented technology. COM and DSOM are designed mainly for document processing and are less emphasized on transaction processing. CORBA, on the other hand, is designed to support general distributed applications.

As shown in Figure 1, CORBA consists of four major components: *object request broker* (ORB), *common object service specification* (COSS), *common facility architecture* (CFA), and application objects. In OMA model, an object that provides service to clients over network is called an *object implementation*. ORB serves as a software interconnection bus between clients and object implementations. COSS defines several commonly used services in distributed systems, such as *Object Transaction Service*, *Persistent Object Service*, etc. CFA specifies a few facilities that are closer to the application level and are more toward to specific application domains, such as common task management tools and facilities for financing and accounting systems. CORBA (Common Object Request Broker Architecture) defines the interfaces and functionality of ORB via which client programs may access services from application servers and/or services provided by COSS and CFA.

The objective of this research is to implement the Object Transaction Service (OTS) defined by the CORBA specification on both Unix and Windows 95 platforms so that distributed transactional applications can be efficiently developed with embedded transactional properties. Furthermore, we wish to implement the OTS with fault-tolerant capability to such an extend that distributed applications can be built to tolerate both software as well as hardware failures.

The basic concept of how a transaction proceeds within the framework of OTS is briefly described below. In a typical scenario, a client first begin a transaction by issuing a request to the OTS manager which establishes a transaction context associated with the client thread. The client then issues transactional operations on the target data objects. A data object (called *resource object*) shall register themselves to the transaction context (maintained by the OTS manager) when it joins the transaction. Eventually, the client ends the transaction by issuing another request to the OTS which coordinates the commit procedure with all objects involved in this transaction on behalf of the client. Thus, the OTS manager is responsible of maintaining the ACID properties [11] of the registered resource objects with respect to a transaction in the presence of

failures.

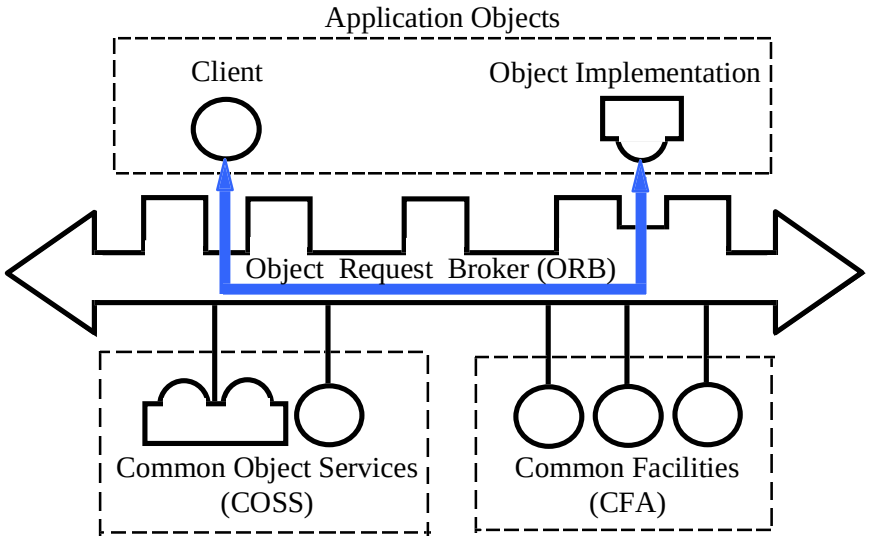


Figure 1. The OMA Framework.

There are many CORBA compliant commercial products available, such as Iona's Orbix [12,13,14], IBM's SOM [8], Sunsoft's NEO [15], and Digital's ObjectBroker [16]. We choose Orbix as our development platform for two reasons; first, it supports both Unix and Window NT platforms, and secondly, it is a more mature products.

The design of the OTS consists of three parts: the *OTS manager* (a run-time daemon), a library and associated header files (used by application clients and application data servers), and the base classes that defines the fundamental transactional operations for application resource objects to inherit. Application programmers can create a resource object (discussed in Section 2) by inheriting those transactional classes offered by the OTS so that it is able to participate transactions in a proper manner. Furthermore, the ACID properties of a resource object is guaranteed by the OTS manager even if failures occur. Transactional applications developed with the support of our OTS implementation are able to tolerate the node crashes, software system failures and process hangs. More over, checkpoint techniques are deployed to ensure the data consistency of transactional objects. The current version of our OTS supports only flat transactions on Unix machines (Sun workstations with Orbix 1.3), no nested transactions are supported. The Window NT version of the OTS shall be ready soon. The preliminary results show that the overhead due to the OTS service is between 15% to 90% depending on the nature of the applications. Possible approaches to reduce the OTS overhead will also be discussed in this paper.

In section 2, we briefly introduce the background of CORBA and the OTS specifications. In section 3, we explore the design of our OTS system architecture. In section 4, we discuss the failure recovery of OTS. In section 5, we discuss the implementation issues of OTS. In section 6, we describe the performance experiment of our work. In section 7, we summarize the contribution of this paper and discuss the future works.

2. Background

In this section, we introduce the basic concepts of CORBA and the Interface Definition Language (IDL). Then, we give an overview of the OTS specifications.

2.1 Application Development in CORBA

As shown in Figure 2, the CORBA environment consists of four major components: the client, the object implementation (or server), the IDL compiler and the ORB. We briefly describe the functionality of each of these components, and then we explore their interactions during run time and the program development phase

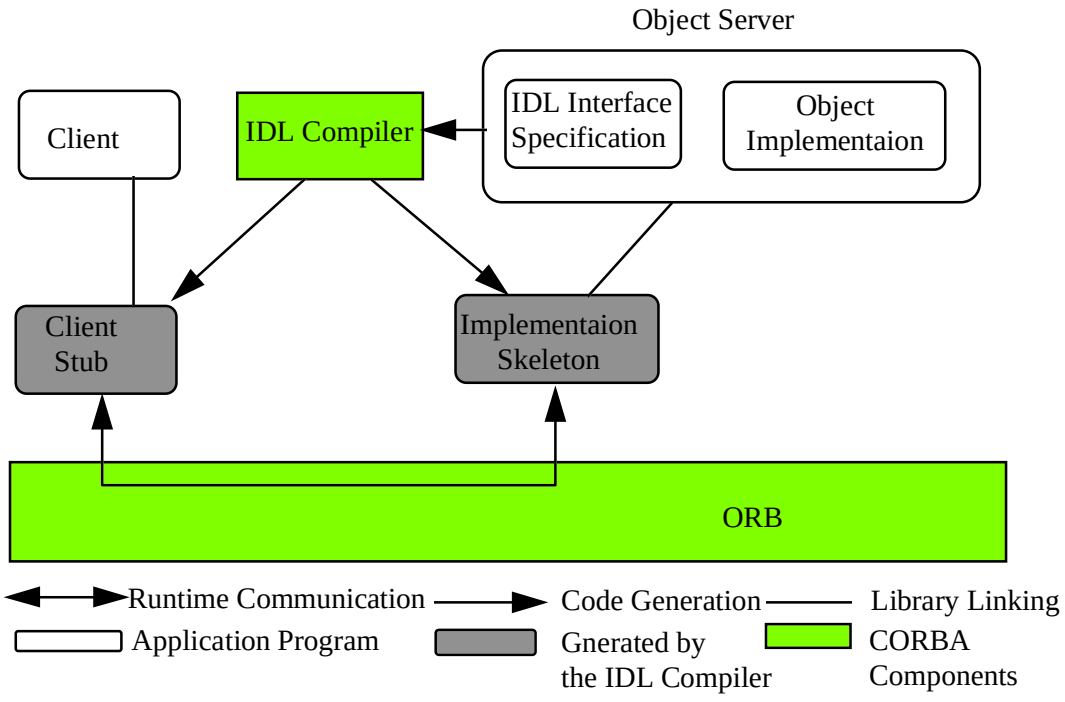


Figure 2. Common Object Request Broker Architecture.

In CORBA, the service of an object server is specified in terms of the application program interface (API) using the standard CORBA interface definition language

(IDL). An object that implements the service (or API) according to an IDL interface specification is called an object implementation (with respect to that IDL interface). In particular, an object implementation is an executable entity that is capable of providing the service specified by the IDL interface to clients over the network. A client makes a request to an object implementation and expects the reply from it via ORB. In this sense, the ORB serves as a software interconnection bus between the client and the object implementation. As shown in Figure 2, a client is able to access the services provided by an object implementation only if it has the handle of that object, i.e., the client stub. The client stub is generated by IDL compiler after it compiles the IDL interface of the object server. On the other hand, an object implementation can provide its services over the network via ORB only if it has the implementation skeleton, which is also generated by the IDL compiler. Figure 3 illustrates the development of the object implementation and the client program.

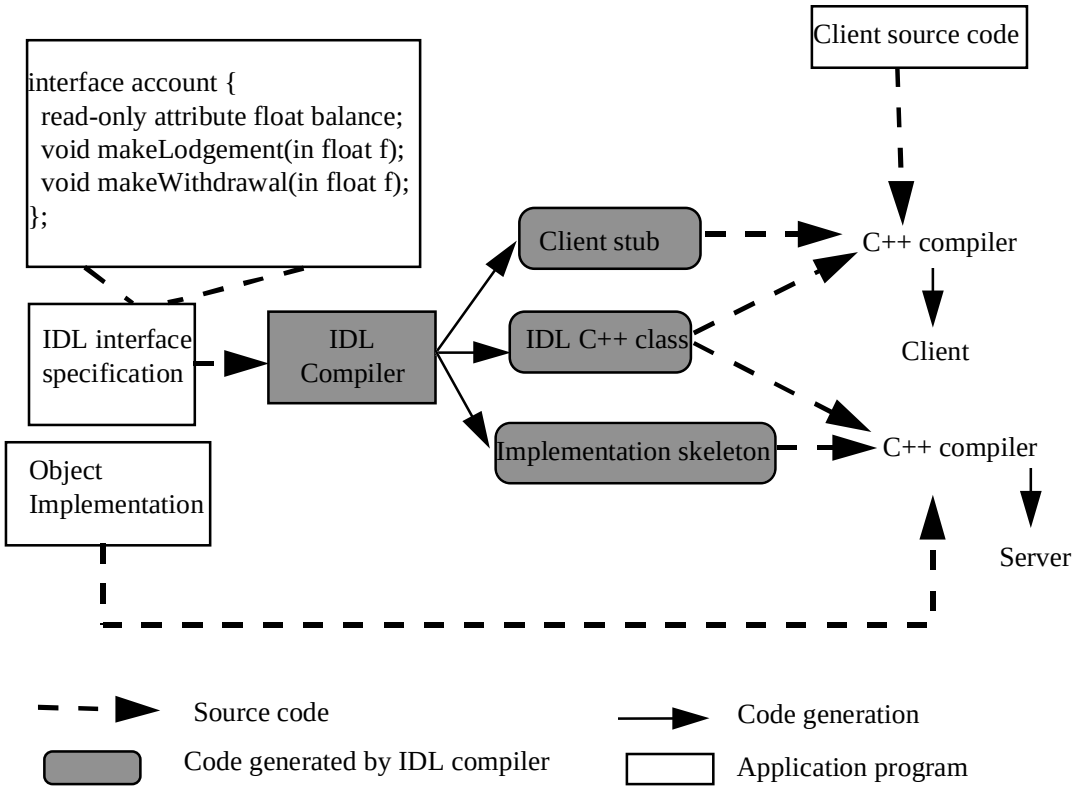


Figure 3. The application development in CORBA.

We now briefly describe the basic concept of how CORBA and its components operate. To invoke an operation on a remote object implementation, a client must first bind to that object. ORB first checks if the remote object implementation exists; if not, a new instance of the object implementation will be invoked. The object implementation replies the results of the invocation to the client via ORB after the object

implementation completes the execution of the invocation. The client may send subsequent requests to the same object implementation without the re-establishment of the communication channel.

Notice that the IDL is a definition language; it is not an operational programming language such as C or C++. Given an IDL interface, there may exist many object implementations for that interface. Moreover, these object implementations may be implemented in various programming languages for different operating systems or hardware platforms. To reduce the coupling between the IDL interface and the object implementation, an IDL compiler must be able to compile the IDL interface then produce the client stub and the implementation skeleton into many target programming languages according to CORBA language bindings, such as C, C++, FORTRAN, PASCAL, etc. We notice that the client stub acts as an local proxy to the client process on behalf of an object implementation that provides the actual service; more precisely, the client stub shields the complex operations, such as remote request preparation, parameters interpretation, request invocation and reply delivery from the client. Similarly, the implementation skeleton acts as the local proxy of a client that makes the request. Finally, we turn our attention to exception handling. ORB raises an exception signal to the client proxy (the client stub) if the peer object implementation crashes or hangs during the request invocation; the client may react to this signal via an exception handling routine. This exception handling mechanism is used to implement the failure recovery process of the OTS implementation.

2.2 Overview of the OTS Specification

This section introduces Object Transaction Service (OTS). As shown in Figure 4, the OMG's specifications for OTS consists of several components. In this section, we briefly describe the functionality of these components and their interrelationship. Interested readers are referred to [1] for detail.

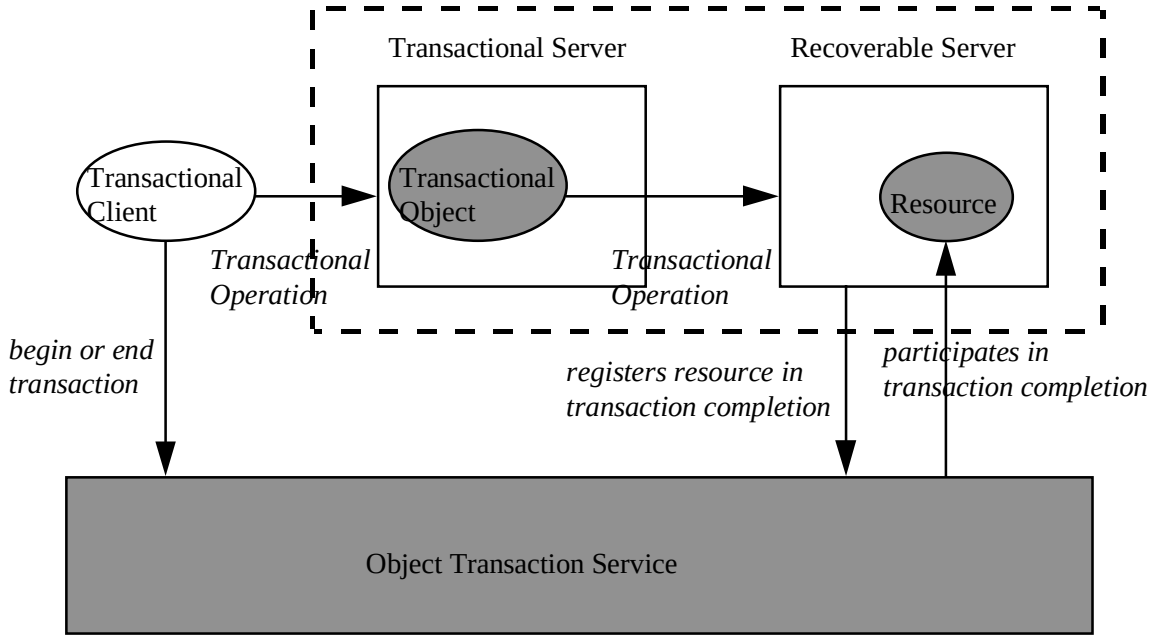


Figure 4. Overview of object transaction service.

Applications supported by OTS consist of the following entities: *Transaction Client* (TC), *Transaction Objects* (TO), *Recoverable Objects* (RO), *Transactional Servers* (TS), and *Recoverable Servers* (RS).

A *transaction client* (TC), also called a transaction originator, is a process that initiates a transaction. To initiate a transaction, a TC first binds to OTS manager (described later) by means of ORB, and then requests the OTS manager to create a new transaction context associated with the client program. After that, the TC may invoke transactional operations to transactional objects (described below). TC is also responsible to initiate the termination process.

A transactional object (TO) is an object whose behavior is influenced by being invoked within the scope of a transaction. It is not necessary that all methods of a transactional object are transactional. A TO can provide both transactional and non-transactional operations. In opposite, an object of which none of the methods is transactional is a *non-transactional* object. Typically, a transactional object contains ("is-a") or indirectly refers to ("has-a") the persistent data that can be modified by requests from the TC. A transactional object receives the transactional operations from the TC and these operations will be forwarded to the ROs.

The object updating data during transaction processing is called recoverable object (RO) for the object has the ability to recover itself in the presence of failures (see Section 4). To complete a transaction, an RO has the responsibility to support the

features defined in the OTS specification. With these features, an RO can cooperate with OTS to ensure that all other participating objects (called *transaction participants*) reach the same decision (commit or rollback) at the end of the transaction, even in the presence of failures. When recoverable objects are invoked by a TC for the first time, they should register themselves to OTS to become participants of the transactions. At the end of the transaction, ROs also involve in the two-phase commit protocol coordinated by OTS. During the processing of the transaction, ROs must store certain information in the persistent storage. As a result, when the recoverable object restarts after failures, it can recover their states and participate in the recovery protocol to properly complete the transaction.

A transactional server (TS) is a collection of transactional objects that have no recoverable states of its own. A transactional server propagates the transaction context to ROs when their methods are invoked. In other words, the transactional server receives the transactional operations from the TC and forwards these operations to the RO. The recoverable server (RS) is a collection of objects and at least one of these objects is recoverable. Notice that in real applications the transactional servers may not exist solely. It is common that the functionality of the TS and the RS are coexisted within the same object implementation.

Next, we introduce the OTS manager. The major functionality of the OTS manager is to cooperate with transaction participants so that a transaction can proceed with ACID properties been enforced. OTS manager is composed of the following functional components: *Factory, Control, Terminator, Coordinator* and *RecoveryCoordinator*. These components are specified with IDL by which other objects may access their services via ORB (see Appendix A). The functionality of these components is illustrated via the following example.

Figure 5 depicts the scenario that a TC cooperates with various functional components of the OTS manager to complete a transaction.

1. The transaction originator begins a new transaction by issuing a request to the Factory and a unique Control object is returned.
2. Through Control, the client can get Coordinator to provide the service throughout the transaction.
3. The transaction originator then begins invoking operations on the ROs (through TOs) with the information of Coordinator as an input parameter.
4. ROs will register themselves to the Coordinator the first time they are invoked within the transaction domain and thus get the Recovery Coordinator.

5. The client uses the Control object to get the Terminator object which is in charge of the termination process of the transaction.
6. The Terminator will coordinate the termination process among Resource objects using a proper commit protocol.

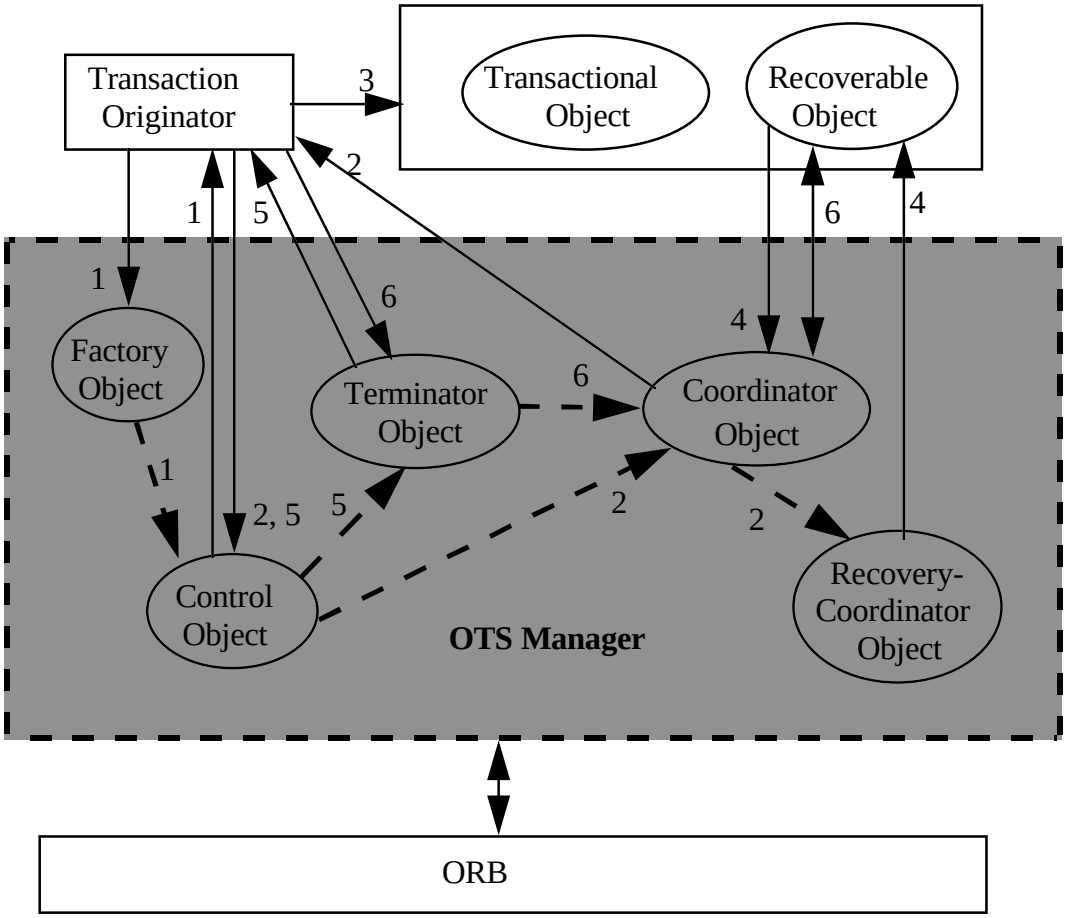


Figure 5. The functional diagram of the applications and the OTS manager.

3. System Architecture

The objective of this research is to implement the Object Transaction Service (OTS) defined by the CORBA specification on both Unix and Windows NT platforms so that distributed transactional applications can be efficiently developed to tolerate both software as well as hardware failures. In this section, we present the design of the current OTS implementation on the platform of Sun Solaris 4.1.3 and Orbix 1.3. The current version of our OTS supports only flat transactions, no nested transactions are supported.

As described in the previous section, the applications supported by the OTS are

TC, TO, and RO. In the following section, we first present the development environment of TC, TO and RO. Then, we present the design of the OTS manager.

3.1 The design of TC and RO

Figure 6 depicts the implementation of the RO and TC. As described in the OTS specification [1], an RO is by definition a TO. In fact, the semantic of ROs is usually rich enough to implement most transactional applications. As a result, we present the development of the RO only. For the implementation of the RO, its IDL definition must inherit the OTS IDL definition. The code of the internal functionality of the RO has to compile with the RO server stub (generated by the IDL compiler) and linked with our OTS library and OTS stub. Next, we introduce the development of a TC. The development of a TC is similar to an RO as depicted in Figure 6. The application program of the TC has to be compiled with the client stubs of those ROs needed to complete the transaction. (Note that those client stubs are generated by the IDL compiler from the IDL of the corresponding ROs.) Furthermore, the TC's application program has to link the OTS stub to produce the TC executable codes.

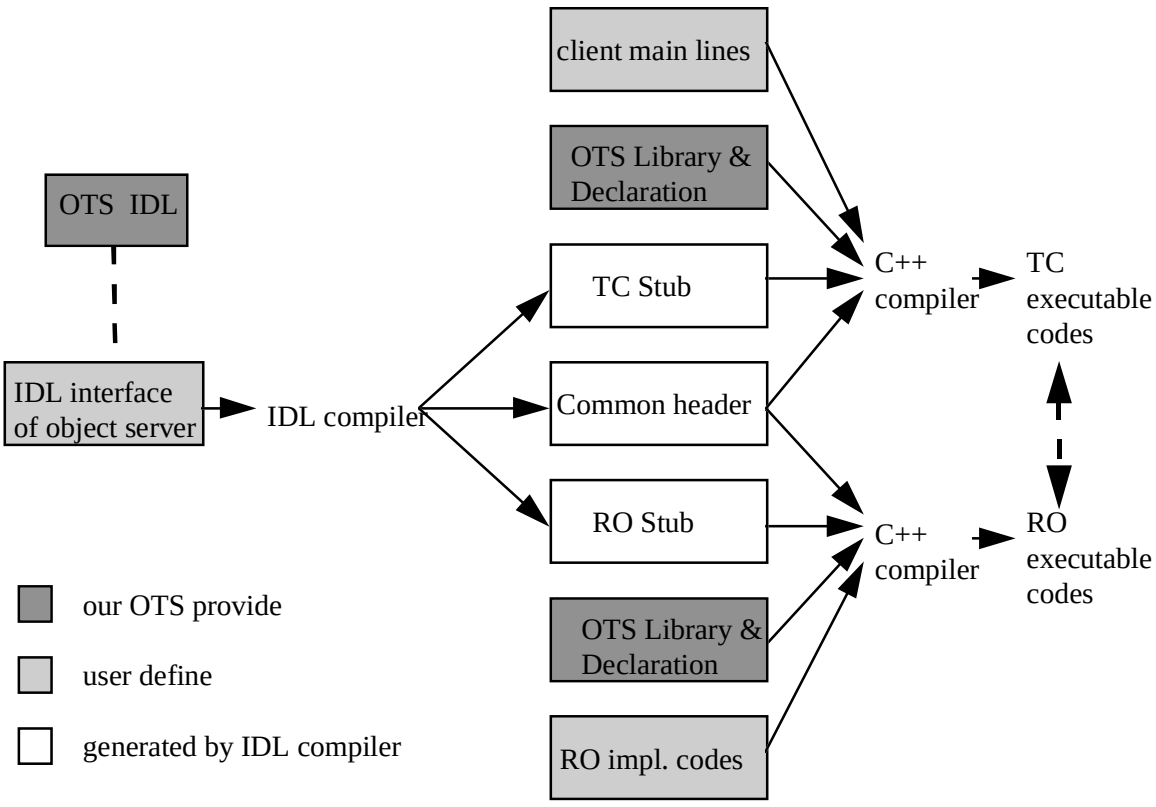


Figure 6. The OTS programming development environment.

3.2 The Design of the OTS manager

The design of the OTS consists of three parts: the *OTS manager* (a run-time daemon), a class that defines the behaviors of the transactional participants required by the OTS for inheritance purpose, and a library and header files (used by users' client and server) . Application programmers can create an RO by inheriting the interfaces offered by the OTS so that it is able to properly participate transactional activities upon the invocations from TCs. (The OTS interfaces specified by the OMG are listed in Appendix A. Interested readers may refer to [1] for details.) A typical life cycle of a transaction consists of three phases: *transaction initialization*, *operation invocations*, and *transaction termination*. We will introduce these interfaces in the order of the life cycle of a typical transaction.

Transaction Initialization

The procedure to initialize a new transaction in OTS is illustrated in Figure 7. In a distributed environment, there may exists multiple OTS managers. A TC first binds to an available OTS manager and then invokes the operation *Factory::create()* to initialize a new transaction. Upon receiving the request, the Factory object creates a Control object and return its object reference to the TC. The Control object defines the transaction context and is responsible for the following processing of that transaction. Notice that the TC may set a time-out value to limit the transaction processing time. If the time-out period expires, the transaction shall roll back. The Control object can be used to create the other two important entities, the Coordinator and the Terminator, via the interfaces: *Control::get_coordinator()* and *Control::get_terminator*(). The functionality of these two objects will be discuss later.

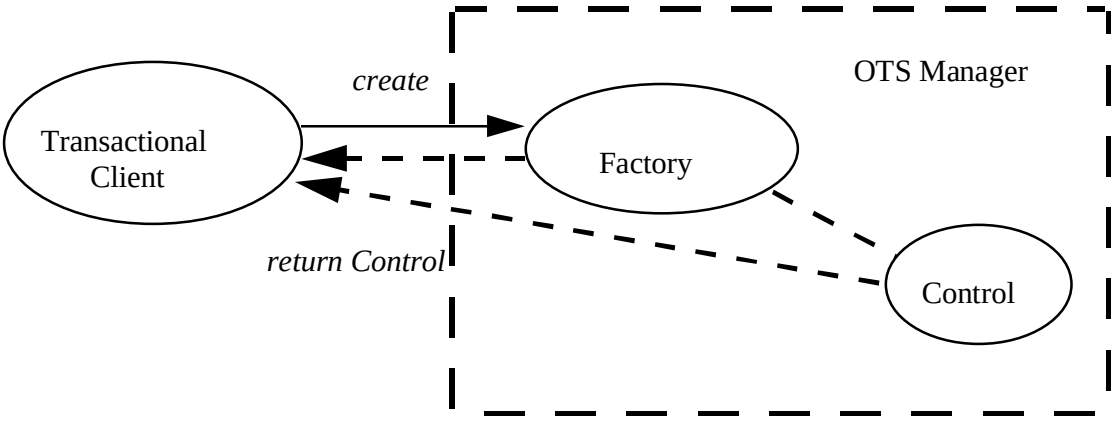


Figure 7. Transaction initialization in OTS.

Operation Invocation

A transaction can involve multiple ROs with multiple requests. All involved ROs (or called transaction participants) share the same transaction context referenced by the Control object. Therefore, we need a mechanism to ensure the ACID properties of all transaction participants upon the completion of that transaction. The Coordinator in OTS serves the purpose. A Coordinator object is responsible for maintaining the transaction context and for governing the coordination among all transaction participants during the commit phase. The TC gets a reference to the Coordinator via the interface *Control::get_coordinator()*. A transaction participant must issue a *Coordinator::register_resource()* operation to the Coordinator object when one of its transactional operations is invoked by the transaction originator for the first time. Because we adopt the explicit mode to propagate the transaction context, all transactional operations of an RO must have an object reference to the Coordinator object as the last parameter.

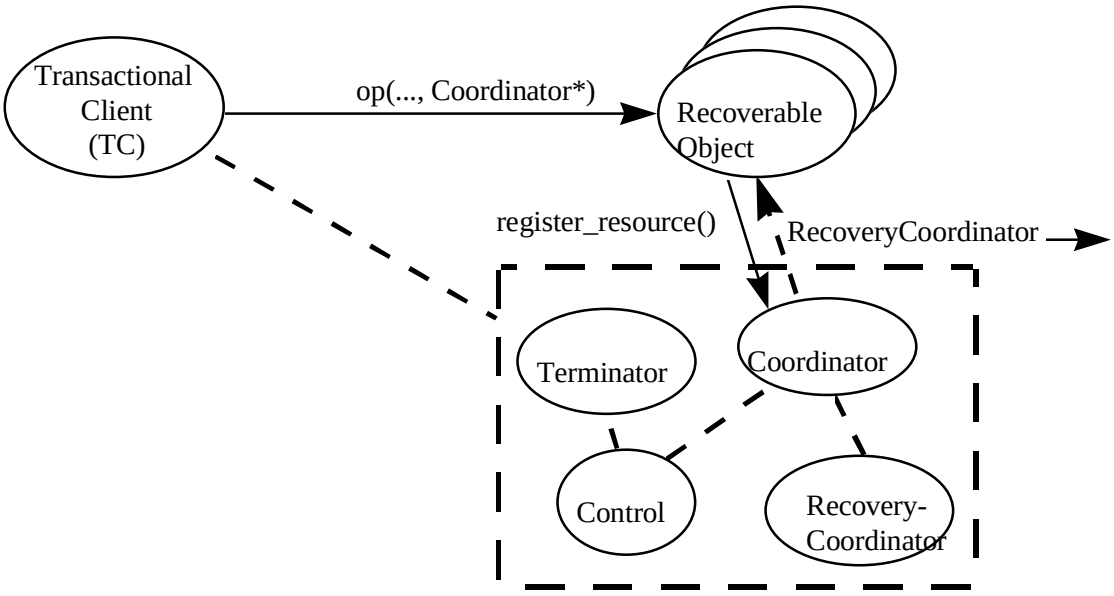


Figure 8. The propagation of transaction context.

Transaction Termination

At the end of the transaction, the TC commits or aborts the transaction via Terminator object. The TC invokes *Control::get_terminator()* and gets a reference to the Terminator object. The TC issues the *Terminator::commit()* operation to mark the

end of the transaction. In some cases, a transaction originator may wish to roll back the transaction, then it uses a *Terminator::rollback*() operation to end the transaction. The TC can invoke Terminator::commit() to start the commit procedure. In this work, two-phase commit protocol (2PC) is implemented as the default commit protocol. It is the TC that issues commit (or abort) request to the transaction and the request is directed to the Coordinator object through the corresponding Terminator Object. Then the Coordinator object communicates with the transaction participants to complete the 2PC. In addition to their own operations, transaction participants must implement transactional behaviors to ensure the ACID properties. To complete a transaction, transaction participants have the responsibilities to: to register themselves to the Coordinator object in the beginning; to participate in 2PC protocol; and to support transaction recovery in case of failures. The ROs must provide the proper APIs via which the Coordinator object may invoke to conduct the commit procedure.

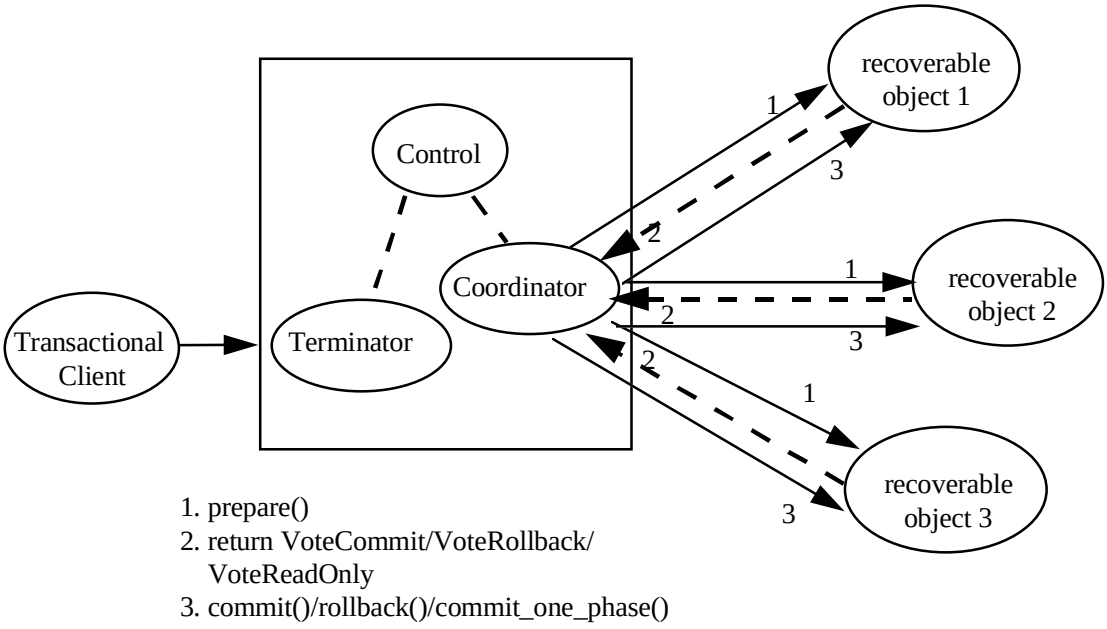


Figure 9. Communication in the Two-phase Commit Protocol.

We now briefly describe the interactions between the Coordinator object and other ROs during the 2PC protocol. Figure 9 depicts the scenario. The processing of two-phase commit protocol can be divided into the following four steps:

1. The Coordinator object invokes the *Resource::prepare()* method on each of the ROs.

2. When a RO's *Resource::prepare()* is invoked, it checks its own state to see if it can commit its part of the transaction, then replies its vote to the Coordinator

object. The vote can be VoteReadOnly, VoteCommit or VoteRollback.

3. The Coordinator object collects the votes from each RO. If any of ROs returns VoteRollback, the Coordinator object decides to roll back the transaction and invokes the *Resource::rollback()* on all ROs which reply VoteCommit. If at least one RO votes VoteCommit and others vote VoteCommit or VoteReadOnly, the Coordinator object commits the transaction by invoking the *Resource::commit()* on each of ROs. If all ROs vote VoteReadOnly, the transaction completes immediately and there is no further operation is required.

4. ROs that vote VoteCommit are waiting for a *Resource::commit()* or a *Resource::rollback()* from the Coordinator object. Each of the ROs must implement its commit and rollback operations, so that they can act accordingly.

There is a special case that only one participant registered to the transaction. The first phase (voting phase) is not necessary here. Instead of issuing *prepare()*, *commit()* or *rollback()* on the single RO, the Coordinator object can invoke *Resource::commit_one_phase()* on it.

4. Failures and Recovery

In this section, we present the design of the failure recovery protocols of OTS. Fault tolerance mechanisms for both the OTS manager and the transaction participants to tolerate the transient failures were designed to ensure the ACID properties of a transaction. We first define a few terminology, and then discuss the failures and the recovery of the OTS manager as well as the transaction participants. Finally, we argue the robustness of the fault-tolerance protocols.

Failure Models

OTS ensures an atomic outcome for transactions even if processes, systems or communication failures occurred. For an object involved in a transaction, we can divide the possible failures into two types, depending on where these failures take place [1]. They are *local failures*, where the failures that affect the object itself, and *external failures*, where the failures that are external to the object. We will consider the failure behaviors and the recovery of different objects in a transaction in both local and external failures in the following discussion.

Recovery Point

To ensure the atomicity of a transaction, we must have some extra controls over

the processing of the 2PC protocol. By recording states in the persistent storage at the proper time, OTS can recover the transaction in case of errors. For efficiency, we adopt the presumed abort strategy with the 2PC protocol. In other words, the objects involved in the transaction defer the message logging until the commit decision is made by the TC. We name the point at which logging takes place as recovery point. When reaching the recovery point, the object has to record the necessary information to the persistent storage. These data will be cleared when the transaction completes successfully.

The recovery of the transaction service can be divided into two categories: the recovery of the Coordinator object (the OTS manager) and the recovery of the Resource object (or the RO). We will describe both types of recovery in the following sections.

If the Coordinator object collects all votes and finds that at least one of these votes is VoteCommit (The other votes can be either VoteCommit or VoteReadOnly), the Coordinator object will make the commit decision and reaches its recovery point. As for the recovery point of the Resource object, it is the time when the Resource object decides to vote VoteCommit to the Coordinator object.

4.1 The Failure Recovery of OTS manager

Local Failure

In this subsection, we are going to discuss the local failure and the recovery of the OTS manager. As shown in Figure 10, the process of a transaction is divided into three phases. There is no additional operation needed for the OTS manager but to roll back the transaction in Phases I and II. For an RO participating a transaction, it inquires the transaction outcome when its time-out period has expired, and it will find that the transaction is rolled back. However, if an error occurs in phase III, the recovery of the OTS manager becomes necessary.

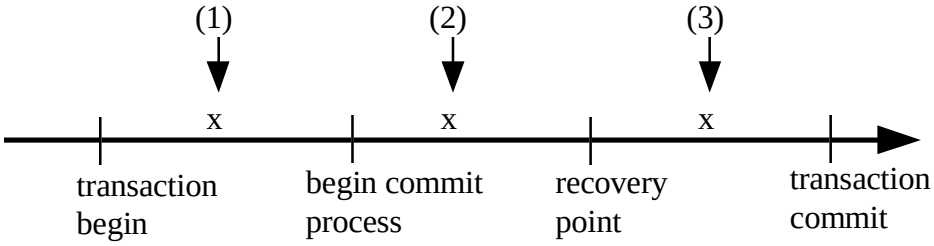
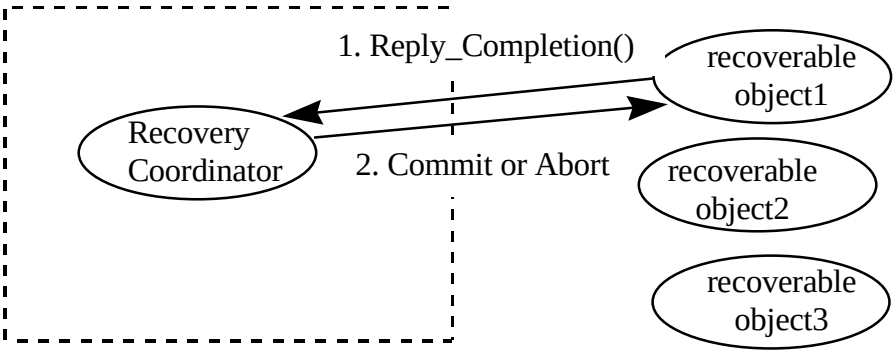


Figure 10. The failure cases of the OTS manager.

Figure 11 shows the recovery of the OTS manager. The RO will issue *RecoveryCoordinator::replay_completion()* to initiate the recovery of the transaction Coordinator. If the OTS manager does fail, this failure is detected by examining a flag raised by the ORB. The failure recovery process of the OTS manager is then initiated by a mechanism provided by Orbix called *Loader*[6]. We will discuss Loader in more detail in Section 5.



1. Replay_Completion()
2. The Recovery Coordinator issues the transaction decision again (Commit() or Rollback())

Figure 11. The recovery of the OTS manager.

External Failure

There may be two possible types of external failures: the participants and the communication errors. From the exception return, programmers can distinguish these two types of failures. If the Coordinator finds that the participant does not exist, it will complete the commitment. If the Coordinator receives an exception that reports the communication failure, it will retain the outcome and try sending it again later.

4.2 The Failure Recovery of the Recoverable Objects

Local Failure

In this section, we discuss the local failure and the recovery of ROs. As shown in Figure 12, the process of a transaction is divided into three phases. In Phases I and II, no recovery is needed since the failure occurs before the recovery point. In these cases, the transaction is forced to roll back when the Coordinator object can not access the RO. However, if the failure occurs in Phase III, that is, the failure occurs after the recovery point, the RO ought to be recovered when the Coordinator issues

commit()/rollback(). The ROs participating in a transaction have the obligation to follow the rules set by OTS.

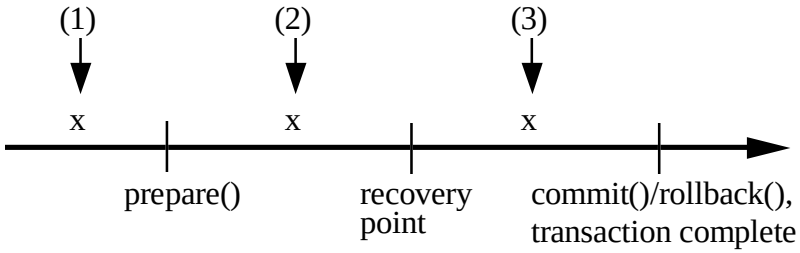


Figure 12. The failure cases of the RO.

External Failure

There are two possible types of external failures: the Coordinator and the communication errors. From the exception return, programmers can distinguish these two types of failures. If an exception indicating that the Coordinator is no longer exists received, i.e., the Coordinator fails before making the commit decision, then the transaction is rolled back. However, if the exception is caused by the communication failure, the prepared Resource will try to complete the transaction later until the communication resumes.

5. Implementation Issues

This section describes some of the implementation details of OTS, including the OTS Manager, Recoverable Object, Object Loader (a unique feature in Orbix), and the concurrency control mechanism in OTS.

The OTS manager

When a TC initiates a new transaction, a set of functional components, including Control, Terminator, Coordinator and RecoveryCoordinator, will be created by the OTS manager to serve this transaction. The benefit of abstracting the OTS manager into several functional components is that it provides different views of the OTS manager. In our implementation, however, the OTS manager is implemented as a single composite object with all functional entities specified in the OTS.

The Coordinator object is the part that drives the 2PC protocol. The 2PC protocol is straight forward if the transaction can commit normally. We list the

algorithm of the 2PC protocol below. It should be noted that all the exception handling codes are omitted for simplicity.

```
begin
        read_only = true
        for ( each node r in the Resource list)
        begin
                vote = r->res->prepare()
                r->vote = vote
                if (vote == VoteCommit)
                        read_only = false
                else if (vote == VoteRollback)
                begin                   /* second phase of 2PC */
                    read_only = false
                    Status = StatusRollback
                    for (each node r in the Resource list has replied VoteCommit)
                                r->res->rollback()
                end
        end
        if (read_only == true)          /* all participants are read only */
                return
        if (store the recoverable data in stable storage successfully)
        begin                           /* second phase of 2PC */
                Status = StatusCommitted
                if (there is only one node r in the Resource list)
                        r->res->commit_one_phase()
                for (each node r in the Resource list)
                        r->res->commit()
      end
      return
end
```

As shown above, each object "r" contains two flags, "*res*" and "*vote*", that represent the object reference of the Resource object and its vote, respectively. The Coordinator object issues *Resource::prepare()* on these Resource objects and records their votes in the vote field of the corresponding nodes.

It is necessary to keep critical information in the persistent storage to ensure completion of a transaction in case that failures occur. The Coordinator can also use these data to continue the 2PC protocol when failures occur. The information that we retain in the persistent storage includes the transaction ID, the status of the current transaction (commit or rollback), the list of the transaction participants.

The Recoverable Object

In OTS, an RO is designed and implemented as an application program.

Although different applications have different design philosophy and their own special requirements, OTS impose certain obligations on the ROs to ensure the proper execution of the transaction. The responsibilities of an ROs can be divided into two categories. The first one is to register the object to the Coordinator to become a participant of the transaction. The other obligation is that the object must support proper commit protocols. In our OTS, the recovery objects are recommended to implement 2PC protocol. Each RO must design proper rules to make the decision about whether to vote commit, rollback, or read-only. When the Coordinator issues prepare() on each Resource, how and what to reply is dependent on the design of the RO. Programmers of ROs also have to set up their own rules about the decision making for voting commit, rollback, or read-only.

If the RO decides to reply VoteRollback or VoteReadOnly, it is not necessary to log anything in the stable storage. The RO can discard the modification and return to its previous consistent states immediately. However, if the RO decides to reply VoteCommit to prepare(), it must stores the object reference of its Coordinator and RecoveryCoordinator to the persistent storage, so that it can complete the transaction even when some failures occur.

The information that an RO must store at its recovery point includes its states (commit or abort) , the object references of the Coordinator and RecoveryCoordinator, and the status that concerned with the concurrency control.

Object Loader

The Loader mechanism provided by Orbix is used to implement our recovery protocol. The basic concept of Loader is described as follows. When an object invocation arrives at a process but the target object does not exist, then Orbix will raise an object fault. Orbix will return an exception to the caller to handle the object fault. In a similar way, we can design a loader for every specific object to detect the object fault and then start the recovery process.

By designing the loader object, we can recover our OTS manager objects if necessary. Figure 13 illustrates the scenario. In (a), the server operates normally. The Factory object accepts a request to create an "A" object. The client gets the object reference to the "A" object and keeps invoking operations on it. In (b), the host fails and restarts later and re-initializes the environment. In (c), when an invocation on "A" object arrives again, the Orbix launches this server but cannot find the "A" object. In the mean time, Orbix raises an object fault, and our loader object will recover the "A" object from the persistent storage and continue the invocation.

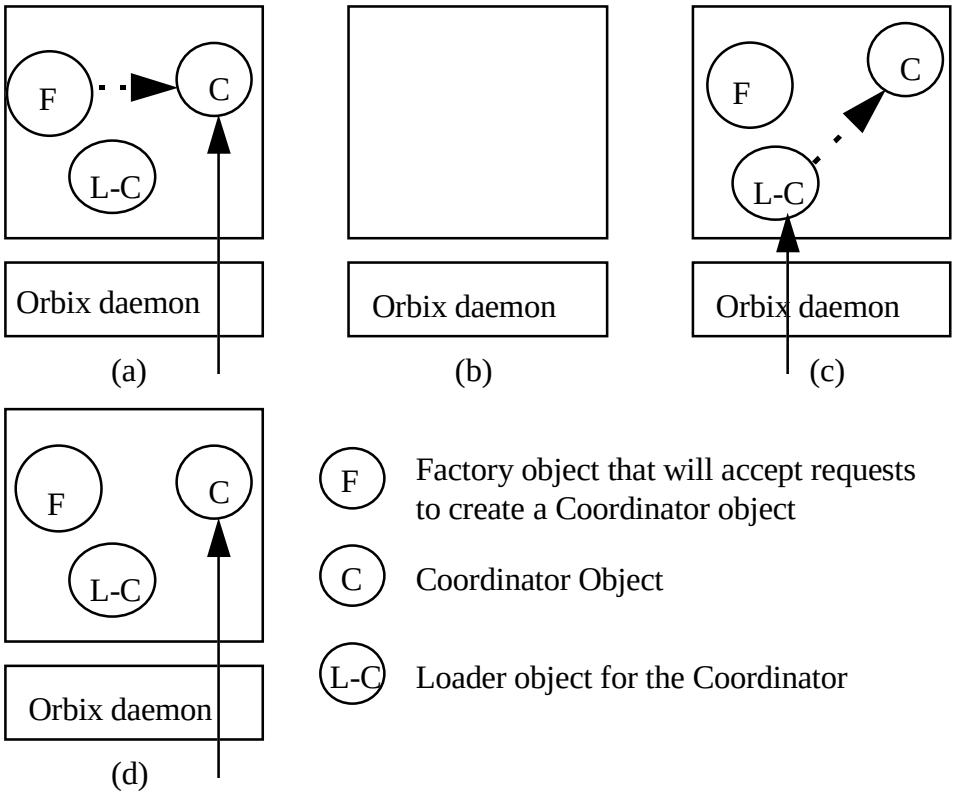


Figure 13. Using Loader Object to Implement the Recovery.

With the Loader mechanism, application programmers can design proper exception handling routines upon the detection of failures. Moreover, the above processes of using the Loader object to implement the recovery is transparent to the client.

Concurrency Control

It is natural that the objects in our system, either the Coordinator object or ROs, will be accessed by different clients at the same time. The current version of Orbix does not support multi-threaded programming, thus the current version of OTS handles one invocation on an RO at one time. If there are multiple requests from different clients on an RO at the same time, OTS buffers these requests and process them one by one. The performance can be a problem if the communication between the server and its clients becomes frequent.

In our implementation, the granularity of the concurrency control is at the object level. Each object has its own private data and public member functions that manipulate its data. These member functions can be more complex than ordinary read and write

operations. A programmer can define its own protocol and embed some control codes in each member function to achieve the concurrency.

The current version of OTS adopts an exclusive lock mechanism *_TxnCheck()*, so that it does not allow the overlap between transactions. Since our concurrency is per-object-based and the size of an object is scaleable, we can decrease the scope of an object to increase the degree of concurrency if necessary.

It will be easy for programmers to choose their desired methods of concurrency control in our OTS implementation, but programmers of the object servers must keep the following issues in mind:

- The concurrency control must support transaction duration locks to protect the data of one transaction from being overwritten by other transactions.
- An RO must store the states related to its concurrency control in the persistent storage when it reaches its recovery point.

6. Performance

In this section, we examine the performance overhead of the OTS via comparing the response time of the applications that use the OTS service with those of the same applications that do not use the OTS service.

6.1 Benchmark Environment

To measure the overhead of our object transaction service, we write an account server that provides both transactional and non-transactional operations. We make our measurement on a Sun ELC workstation that runs SunOS 4.1.3. For the purpose of testing the overhead of a transactional, the operations include the following:

1. The client connects to the OTS manager daemon to initiate a new transaction.
2. The client invokes the transactional operations of the account server, and accounts register themselves to the OTS manager.
3. The client commits the transaction, and the Coordinator initiates the 2PC to complete the transaction.

However, for the purpose of comparison, the client also invokes the same set of operations to ROs in an environment without OTS support.

The IDL interface of the account object is listed below:

```
interface account : Transactions::Resource {
    float     x_deposit(in float amt, in Transactions::Coordinator co)
    float     deposit(in float amt);
    float     x_withdrawal(in float amt, in Transactions::Coordinator co)
    float     withdrawal(in float amt);
    float     x_getBalance(in Transactions::Coordinator co)
    float     getBalance();
};
```

Notice that in the above program listing, *x_deposit()*, *x_withdraw()* and *x_getBalance()* are transactional operations, whereas *deposit()*, *withdraw()* and *getBalance()* are non-transactional operations. We use one client to invoke a set of transactional operations within a transaction, and use another client to invoke the same but non-transactional operations to compare the elapsed time they take.

6.2 Performance Analysis

The overhead evaluation of our object transaction service is carried out in several different cases. We measure the results of issuing 10, 50 and 100 transactional or non-transactional operations to account objects, and the numbers of account servers are 1, 5 and 10 respectively. The results are illustrated in Table 1.

Total no. of invocations / Overhead % / Total no. of servers	10 invocations	50 invocations	100 invocations
1 server	98%	90%	78%
5 servers	73%	65%	61%
10 servers	63%	60%	61%

Table 1. The Result of Overhead Evaluation.

From Table 1, we notice that the major factors that influence the overhead are, the number of the extra message exchange, the operations that store recoverable data to

the persistent storage, and the concurrency control behavior in every transactional operations.

The time used to activate factory server and account servers are excluded in both transactional and non-transactional cases. From these results, we can find that the more the number of operations, the less the overhead is when issuing operations to a fixed number of account servers. In addition, when issuing 10 operations to one server, 50 operations to five servers and 100 operations to ten servers, each server takes 10 operations. However, from the experiment, we find that the more numbers of the servers are, the less the overhead is.

In a transaction, the client needs to communicate with the factory server to create a new transaction, and the Coordinator also needs to issue prepare() and commit()/rollback() to the object servers. Furthermore, during the processing of a transaction, the Coordinator object and Recoverable Servers (the account server in this case) need to store their states in the persistent storage at certain critical time. This is the major part of the overhead.

In our testing cases, the Recoverable Servers (the *account* objects) store their balance in the persistent storage. However, the real-world scenario, the information that needs to be stored in the persistent storage will be much more complex. As a result, the overhead will be higher in a normal application.

For the transaction Coordinator, the information that needs to be stored depends on the number of Recoverable Servers involved in the transaction. However, for a Recoverable Server, the information that needs to be stored in the persistent storage depends on the attributes of the server objects.

As discussed in the previous section, each transactional operation has to perform some operations to check if this object has already involved in a transaction and if the transaction context associated with the operation is the same as the transaction in which this RO participates. These operations are also the overhead that comes with our OTS.

In summary, the overhead to develop transactional applications using our OTS implementation is 60% to 90% as opposed to traditional application without transactional support. The amount of overhead depends on the following factors: the number of transactional invocations, the number of Recoverable Servers involved in the transaction and the amount of recoverable states needed to be stored in the persistence storage.

7. Conclusion

The concept of transactions is not only useful in database applications, but also useful in building robust software for 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. The *Object Transaction Service* (OTS) specification advocated by OMG's CORBA standard defines the fundamental transaction service to support rapid development of transactional applications in a distributed object environment.

This paper presents an implementation of the Object Transaction Service (OTS) based on the CORBA specification. Two types of applications are supported by the OTS: the *transactional clients* (that initiate transactions) and the *ROs* (that maintain the consistence of the data objects accessed and modified by transactional clients during the execution of transactions.) The OTS implementation itself is composed of three components: the OTS manager, the OTS library and the virtual class declarations for the development of ROs. The development of the transactional clients and ROs and their interaction with the OTS manager are also discussed in this paper.

Transactional applications developed with the support of our OTS implementation are able to maintain the ACID properties of the data objects even in the presence of node crashes, software system failures and process hangs. The current OTS implementation support the Unix platform (Solaris 4.1) using a CORBA compliant environment, ORBIX 1.3. This version of OTS supports only flat transactions. The preliminary results obtained from the experiments on Sun workstations with Orbix 1.3 show that the overhead due to the OTS service is 60% to 90% depends on the specific applications.

The next version of our OTS implementation shall focus on several areas. Firstly, nested transaction shall be supported. More platforms, such as Window NT and Window'95, in particular, will also be supported. Furthermore, we wish to integrate the OTS with other common object services such as the *Concurrency Control Service* and *Persistent Object Service* [3] to provide a more comprehensive environment for application programmers.

References

[1] *Object Transaction Service*, Object Management Group, Inc., August 1994.

[2] E. Cobb, "Object and Transactions: Together at last", *OBJECT Magazine*, pp.59-63, January 1995.

[3] R. Orfali and D. Harkey, "Client/Server with Distributed Objects", *BYTE Magazine*, pp.151-162, April 1995.

[4] S. Andrade, M.T. Carges, and K.R. Kovach, "Building a Transaction Processing System on Unix Systems", *UniForum Conference Proceedings*, February 1989.

[5] A. Dwyer, "Enterprise Transaction Processing", *UniForum Conference Proceedings*, January 1991.

[6] A. D. Wolfe, "Transaction Encina", *Distributed Computing Monitor*, vol. 7, no. 11, November 1992

[7] T. R. Halfhill and S. Salamore, "Components Everywhere," *Byte*, vol. 21, No.1, pp. 97-104, January 1996.

[8] IBM, "SOMobjects: A Practical Introduction to SOM and DSOM", *International Technical Support Organization*, July 1994.

[9] John R. Rymer, "IBM's System Object Model", *Distributed Computing Monitor*, vol. 8, No. 3, pp.1-24, March 1993.

[10] *The Common Object Request Broker: Architecture and Specification*, Object Management Group, Inc., December 1993.

[11] T. Harder and A. Reuters, "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, vol. 15, no. 4, 1983.

[12] *Orbix Programmer's Guide*, IONA Technologies Ltd., November 1994.

[13] *Orbix Advanced Programmer's Guide*, IONA Technologies Ltd., November 1994.

[14] *IonaSphere Issue 11*, IONA Technologies Ltd., May 1995.

[15] *NEO Programming Guide*, Beta Version, SunSoft, Inc., May 1995.

[16] "Digital's ObjectBroker--advanced integration of distributed resources," *Aberdeen Group Profile*, November 1995.

[17] F. Pons and J. F. Vilarem, "A Dynamic and Integrated Concurrency Control for

Distributed Databases", *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 3, pp. 364-374, April 1989.

[18] Hamilton, M. L. Powell and J. G. Mitchell, "Subcontract: A flexible base for distributed programming", *Operating System Review*, pp. 69-79, December 1993.

Appendix: IDL specification of OTS

Here is the subset of IDL interfaces defined in the OMG's "Object Transaction Service" specification. The pseudo object Current, Transactional Object, operations about nested transaction and heuristic exceptions are not included in our OTS.

```
// DATATYPES
enum Status {
StatusActive,
StatusMarkedRollback,
StatusPrepared,
StatusCommitted,
StatusRolledBack,
StatusUnknown,
StatusNoTransaction
};

enum Vote {
VoteCommit,
VoteRollback,
VoteReadOnly
};

interface Factory {
Control create(in unsigned long time_out);
};

interface Control {
Terminator get_terminator();
Coordinator get_coordinator();
};

interface Terminator {
void commit(in boolean report_heuristics);
void rollback();
```

```
};

interface Coordinator {
RecoveryCoordinator register_resource(in Resource r);
void rollback_only();
string get_transaction_name();
};

interface RecoveryCoordinator {
Status replay_completion(in Resource r);
};

interface Resource {
Vote prepare();
void rollback();
void commit();
void commit_one_phase();
void forget();
};
```