Transaction Service Specification

September 2002
Version 1.3
formal/02-08-07

An Adopted Specification of the Object Management Group, Inc.

Copyright © 1997 BEA Systems
Copyright © 1994, 1995, 1996 Groupe Bull
Copyright © 1994, 1995, 1996 IBM
Copyright © 1994, 1995, 1996 ICL plc
Copyright © 1994, 1995, 1996 Iona Technologies Ltd.
Copyright © 1994, 1995, 1996 Novell, Inc.
Copyright © 2001, Object Management Group, Inc.
Copyright © 1995, 1996 Sun Microsystems, Inc.
Copyright © 1994, 1995, 1996 SunSoft, Inc.
Copyright © 1994, 1995, 1996 Tandem Computers, Inc.
Copyright © 1994, 1995, 1996 Tivoli Systems, Inc.
Copyright © 1994, 1995, 1996 Transarc Corporation

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

Contents

Preface . . . . **v**

1. Overview **1-1**

1.1 Introduction 1-1

1.2 Service Description 1-2

1.2.1 Overview of Transactions 1-2

1.2.2 Transactional Applications 1-3

1.2.3 Definitions 1-3

1.2.4 Transaction Service Functionality 1-6

1.2.5 Principles of Function, Design, and Performance 1-8

1.3 Service Architecture 1-12

1.3.1 Typical Usage 1-13

1.3.2 Transaction Context 1-13

1.3.3 Context Management 1-14

1.3.4 Datatypes 1-14

1.3.5 Structures 1-16

1.3.6 Exceptions 1-16

2. Transaction Service Interfaces **2-1**

2.1 Introduction 2-2

2.2 Current Interface 2-2

2.2.1 begin 2-3

2.2.2 commit 2-3

2.2.3 rollback 2-3

2.2.4 rollback_only 2-4

2.2.5 get_status 2-4

Contents

- 2.2.6 get_transaction_name 2-4
- 2.2.7 set_timeout 2-4
- 2.2.8 get_timeout. 2-5
- 2.2.9 get_control 2-5
- 2.2.10 suspend. 2-5
- 2.2.11 resume 2-5

2.3 TransactionFactory Interface 2-5
- 2.3.1 create 2-6
- 2.3.2 recreate 2-6

2.4 Control Interface 2-6
- 2.4.1 get_terminator 2-7
- 2.4.2 get_coordinator. 2-7

2.5 Terminator Interface. 2-7
- 2.5.1 commit 2-8
- 2.5.2 rollback. 2-8

2.6 Coordinator Interface 2-8
- 2.6.1 get_status 2-9
- 2.6.2 get_parent_status 2-10
- 2.6.3 get_top_level_status 2-10
- 2.6.4 is_same_transaction 2-10
- 2.6.5 is_ancestor_transaction. 2-11
- 2.6.6 is_descendant_transaction 2-11
- 2.6.7 is_related_transaction. 2-11
- 2.6.8 is_top_level_transaction 2-11
- 2.6.9 hash_transaction 2-11
- 2.6.10 hash_top_level_tran 2-11
- 2.6.11 register_resource 2-11
- 2.6.12 register_synchronization. 2-12
- 2.6.13 register_subtran_aware 2-12
- 2.6.14 rollback_only 2-13
- 2.6.15 get_transaction_name 2-13
- 2.6.16 create_subtransaction 2-13
- 2.6.17 get_txcontext 2-13

2.7 Recovery Coordinator Interface 2-13
- 2.7.1 replay_completion 2-14

2.8 Resource Interface 2-14
- 2.8.1 prepare 2-14
- 2.8.2 rollback. 2-15
- 2.8.3 commit 2-15
- 2.8.4 commit_one_phase 2-16

Contents

2.8.5 forget 2-16

2.9 Synchronization Interface 2-16

2.9.1 before_completion 2-17

2.9.2 after_completion. 2-17

2.10 Subtransaction Aware Resource Interface 2-17

2.10.1 commit_subtransaction. 2-18

2.10.2 rollback_subtransaction 2-18

2.11 TransactionalObject Interface 2-18

2.12 Policy Interfaces. 2-18

2.12.1 Creating Transactional Object References 2-23

2.13 The User’s View. 2-26

2.13.1 Application Programming Models 2-26

2.13.2 Interfaces 2-28

2.13.3 Checked Transaction Behavior 2-28

2.13.4 X/Open Checked Transactions 2-29

2.13.5 Implementing a Transactional Client: Heuristic Completions 2-30

2.13.6 Implementing a Recoverable Server. 2-30

2.13.7 Application Portability 2-31

2.13.8 Distributed Transactions. 2-32

2.13.9 Applications Using Both Checked and Unchecked Services 2-32

2.13.10 Examples 2-32

2.13.11 Model Interoperability 2-35

2.13.12 Failure Models 2-38

2.14 The Implementers’ View 2-40

2.14.1 Transaction Service Protocols 2-40

2.14.2 ORB/TS Implementation Considerations 2-51

2.14.3 Model Interoperability 2-63

Appendix A - Complete OMG IDL A-1

Appendix B - Relationship to TP Standards B-1

Appendix C - Conformance Requirements C-1

Glossary 1

Index. 1

Reference Sheet 1

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*

- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services,* a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities,* a collection of specifications for OMG's Common Facilities. See the individual facility specifications.

- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Healthcare*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
  - *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
  - *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Service Design Principles

Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to "fine-grain" objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as "building blocks."

Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer*, *PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service.
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms.

Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

Interface Style Consistency

Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

Explicit Versus Implicit Operations

Operations are always explicit rather than implied (e.g., by a flag passed as a parameter value to some "umbrella" operation). In other words, there is always a distinct operation corresponding to each distinct function of a service.

Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by "normal" clients.

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- BEA Systems
- Groupe Bull
- IBM
- ICL plc
- Iona Technologies Ltd.
- Novell, Inc.
- SunSoft, Inc.
- Tandem Computers, Inc.
- Tivoli Systems, Inc.
- Transarc Corporation

Overview 1

Contents

This chapter contains the following topics.

Topic	Page
"Introduction"	1-1
"Service Description"	1-2
"Service Architecture"	1-12

1.1 Introduction

This chapter provides the following information about the Transaction Service:

- A description of the service, which explains the functional design and performance requirements that are satisfied by this specification.
- An overview of the Transaction Service that introduces the concepts used throughout this chapter.
- A description of the Transaction Service's architecture and a detailed definition of the Transaction Service, including definitions of its interfaces and operations.
- A user's view of the Transaction Service as seen by the application programmer, including client and object implementer.
- An implementer's view of the Transaction Service, which will interest Transaction Service and ORB providers.

This specification also contains an appendix that explains the relationship between the Transaction Service and TP standards, and a glossary that contains transaction terms.

1.2 Service Description

The concept of transactions is an important programming paradigm for simplifying the construction of reliable and available applications, especially those that require concurrent access to shared data. The transaction concept was first deployed in commercial operational applications where it was used to protect data in centralized databases. More recently, the transaction concept has been extended to the broader context of distributed computation. Today it is widely accepted that transactions are the key to constructing reliable distributed applications.

The Transaction Service described in this specification brings the transaction paradigm, essential to developing reliable distributed applications, and the object paradigm, key to productivity and quality in application development, together to address the business problems of commercial transaction processing.

1.2.1 Overview of Transactions

The Transaction Service supports the concept of a transaction. A transaction is a unit of work that has the following (ACID) characteristics:

- A transaction is **atomic**; if interrupted by failure, all effects are undone (rolled back).
- A transaction produces **consistent** results; the effects of a transaction preserve invariant properties.
- A transaction is **isolated**; its intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they are performed concurrently.
- A transaction is **durable**; the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure).

A transaction can be terminated in two ways: the transaction is either committed or rolled back. When a transaction is committed, all changes made by the associated requests are made permanent. When a transaction is rolled back, all changes made by the associated requests are undone.

The Transaction Service defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity. These interfaces enable the objects to either commit all changes together or to rollback all changes together, even in the presence of (noncatastrophic) failure. No requirements are placed on the objects other than those defined by the Transaction Service interfaces.

Transaction semantics can be defined as part of any object that provides ACID properties. Examples are ODBMSs and persistent objects. The value of a separate transaction service is that it allows:

- Transactions to include multiple, separately defined, ACID objects.
- The possibility of transactions that include objects and resources from the non-object world.

1.2.2 Transactional Applications

The Transaction Service provides transaction synchronization across the elements of a distributed client/server application.

A transaction can involve multiple objects performing multiple requests. The scope of a transaction is defined by a transaction context that is shared by the participating objects. The Transaction Service places no constraints on the number of objects involved, the topology of the application, or the way in which the application is distributed across a network.

In a typical scenario, a client first begins a transaction (by issuing a request to an object defined by the Transaction Service), which establishes a transaction context associated with the client thread. The client then issues requests. These requests are implicitly associated with the client's transaction; they share the client's transaction context. Eventually, the client decides to end the transaction (by issuing another request). If there were no failures, the changes produced as a consequence of the client's requests would then be committed; otherwise, the changes would be rolled back.

In this scenario, the transaction context is transmitted implicitly to the objects, without direct client intervention (see Section 2.2.1, "Application Programming Models," on page 2-25. The Transaction Service also supports scenarios where the client directly controls the propagation of the transaction context. For example, a client can pass the transaction context to an object as an explicit parameter in a request. An implementation of the Transaction Service might limit the client's ability to explicitly propagate the transaction context, in order to guarantee transaction integrity (see Section 2.2.1, "Application Programming Models," on page 2-25).

The Transaction Service does not require that all requests be performed within the scope of a transaction. A request issued outside the scope of a transaction has no associated transaction context. It is up to each object to determine its behavior when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

1.2.3 Definitions

Applications supported by the Transaction Service consist of the following entities:

- Transactional Client (TC)
- Transactional Objects (TO)
- Recoverable Objects
- Transactional Servers
- Recoverable Servers

Figure 1-1 shows a simple application that includes these basic elements.

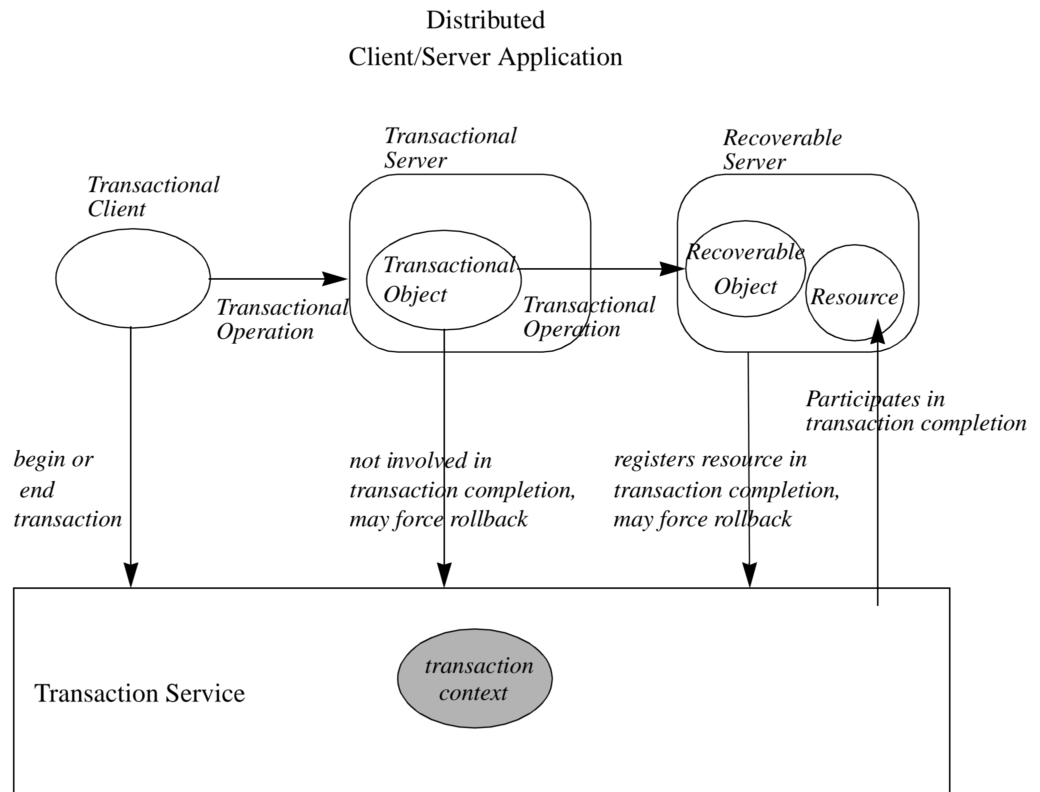

Figure 1-1 Application Including Basic Elements

1.2.3.1 Transactional Client

A transactional client is an arbitrary program that can invoke operations of many transactional objects in a single transaction. The program that begins a transaction is called the transaction originator.

1.2.3.2 Transactional Object

We use the term *transactional object* to refer to an object whose behavior is affected by being invoked within the scope of a transaction. A transactional object typically contains or indirectly refers to persistent data that can be modified by requests.

The Transaction Service does not require that all requests have transactional behavior, even when issued within the scope of a transaction. An object can choose to not support transactional behavior, or to support transactional behavior for some requests but not others.

We use the term *nontransactional object* to refer to an object none of whose operations are affected by being invoked within the scope of a transaction.

If an object does not support transactional behavior for a request, then the changes produced by the request might not survive a failure and the changes will not be undone if the transaction associated with the request is rolled back.

An object can also choose to support transactional behavior for some requests but not others. This choice can be exercised by both the client and the server of the request.

The Transaction Service permits an interface to have both transactional and nontransactional implementations. No IDL extensions are introduced to specify whether or not an operation has transactional behavior. Transactional behavior can be a quality of service that differs in different implementations.

Transactional objects are used to implement two types of application servers:

- Transactional Server
- Recoverable Server

1.2.3.3 Recoverable Objects and Resource Objects

To implement transactional behavior, an object must participate in certain protocols defined by the Transaction Service. These protocols are used to ensure that all participants in the transaction agree on the outcome (commit or rollback) and to recover from failures.

To be more precise, an object is required to participate in these protocols only if it directly manages data whose state is subject to change within a transaction. An object whose data is affected by committing or rolling back a transaction is called a recoverable object.

A recoverable object is by definition a transactional object. However, an object can be transactional but not recoverable by implementing its state using some other (recoverable) object. A client is concerned only that an object is transactional; a client cannot tell whether a transactional object is or is not a recoverable object.

A recoverable object must participate in the Transaction Service protocols. It does so by registering an object called a *Resource* with the Transaction Service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction.

A recoverable object typically involves itself in a transaction because it is required to retain in stable storage certain information at critical times in its processing. When a recoverable object restarts after a failure, it participates in a recovery protocol based on the contents (or lack of contents) of its stable storage.

A transaction can be used to coordinate non-durable activities that do not require permanent changes to storage.

1.2.3.4 Transactional Server

A transactional server is a collection of one or more objects whose behavior is affected by the transaction, but have no recoverable states of their own. Instead, it implements transactional changes using other recoverable objects. A transactional server does not participate in the completion of the transaction, but it can force the transaction to be rolled back.

1.2.3.5 Recoverable Server

A recoverable server is a collection of objects, at least one of which is recoverable. A recoverable server participates in the protocols by registering one or more *Resource* objects with the Transaction Service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction.

1.2.4 Transaction Service Functionality

The Transaction Service provides operations to:

- Control the scope and duration of a transaction.
- Allow multiple objects to be involved in a single, atomic transaction.
- Allow objects to associate changes in their internal state with a transaction.
- Coordinate the completion of transactions.

1.2.4.1 Transaction Models

The Transaction Service supports two distributed transaction models: flat transactions and nested transactions. An implementation of the Transaction Service is not required to support nested transactions.

Flat Transactions

The Transaction Service defines support for a flat transaction model. The definition of the function provided, and the commitment protocols used, is modeled on the X/Open DTP transaction model definition.[1]

A flat transaction is considered to be a top-level transaction that cannot have a child transaction.

---

1. See *Distributed Transaction Processing: The XA Specification,* X/Open Document C193. X/Open Company Ltd., Reading, U.K., ISBN 1-85912-057-1.

Nested Transactions

The Transaction Service also defines a nested transaction model. Nested transactions provide for a finer granularity of recovery than flat transactions. The effect of failures that require rollback can be limited so that unaffected parts of the transaction need not rollback.

Nested transactions allow an application to create a transaction that is embedded in an existing transaction. The existing transaction is called the *parent* of the subtransaction; the subtransaction is called a *child* of the parent transaction.

Multiple subtransactions can be embedded in the same parent transaction. The children of one parent are called *siblings*.

Subtransactions can be embedded in other subtransactions to any level of nesting. The *ancestors* of a transaction are the parent of the subtransaction and (recursively) the parents of its ancestors. The *descendants* of a transaction are the children of the transaction and (recursively) the children of its descendants.

A top-level transaction is one with no parent. A top-level transaction and all of its descendants are called a *transaction family.*

A subtransaction is similar to a top-level transaction in 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 of the transaction's ancestors.

Subtransactions are strictly nested. A transaction cannot commit unless all of its children have completed. When a transaction is rolled back, all of its children are rolled back.

Objects that participate in transactions must support isolation of transactions. The concept of isolation applies to subtransactions as well as to top level transactions. When a transaction has multiple children, the children appear to other transactions to execute serially, even if they are performed concurrently.

Subtransactions can be used to isolate failures. If an operation performed within a subtransaction fails, only the subtransaction is rolled back. The parent transaction has the opportunity to correct or compensate for the problem and complete its operation. Subtransactions can also be used to perform suboperations of a transaction in parallel, without the risk of inconsistent results.

1.2.4.2 Transaction Termination

A transaction is terminated by issuing a request to commit or rollback the transaction. Typically, a transaction is terminated by the client that originated the transaction—the transaction originator. Some implementations of the Transaction Service may allow transactions to be terminated by Transaction Service clients other than the one that created the transaction.

Any participant in a transaction can force the transaction to be rolled back (eventually). If a transaction is rolled back, all participants rollback their changes. Typically, a participant may request the rollback of the current transaction after encountering a failure. It is implementation-specific whether the Transaction Service itself monitors the participants in a transaction for failures or inactivity.

1.2.4.3 Transaction Integrity

Some implementations of the Transaction Service impose constraints on the use of the Transaction Service interfaces in order to guarantee integrity equivalent to that provided by the interfaces, which support the X/Open DTP transaction model. This is called *checked* transaction behavior.

For example, allowing a transaction to commit before all computations acting on behalf of the transaction have completed can lead to a loss of data integrity. Checked implementations of the Transaction Service will prevent premature commitment of a transaction.

Other implementations of the Transaction Service may rely completely on the application to provide transaction integrity. This is called *unchecked* transaction behavior.

1.2.4.4 Transaction Context

As part of the environment of each ORB-aware thread, the ORB maintains a transaction context. The transaction context associated with a thread is either null (indicating that the thread has no associated transaction) or it refers to a specific transaction. It is permitted for multiple threads to be associated with the same transaction at the same time, in the same execution environment or in multiple execution environments.

The transaction context can be implicitly transmitted to transactional objects as part of a transactional operation invocation. The Transaction Service also allows programmers to pass a transaction context as an explicit parameter of a request.

1.2.4.5 Synchronization

The Transaction Service defines support for a synchronization interface. This provides a protocol by which an object may be notified prior to the start of the two-phase commit protocol within the coordinator with which it is registered. An implementation of the Transaction Service is not required to support synchronization.

1.2.5 Principles of Function, Design, and Performance

The Transaction Service defined in this specification fulfills a number of functional, design, and performance requirements.

1.2.5.1 Functional Requirements

The Transaction Service defined in this specification addresses the following functional requirements:

Support for multiple transaction models. The flat transaction model, which is widely supported in the industry today, is a mandatory component of this specification. The nested transaction model, which provides finer granularity isolation and facilitates object reuse in a transactional environment, is an optional component of this specification.

Evolutionary Deployment. An important property of object technology is the ability to “wrapper” existing programs (coarse grain objects) to allow these functions to serve as building blocks for new business applications. This technique has been successfully used to marry object-oriented end-user interfaces with commercial business logic implemented using classical procedural techniques.

It can similarly be used to encapsulate the large body of existing business software on legacy environments and leverage that in building new business applications. This will allow customers to gradually deploy object technology into their existing environments, without having to reimplement all existing business functions.

Model Interoperability. Customers desire the capability to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires that a single transaction be shared by both the object and procedural code. This includes the following:

- A single transaction that includes ORB and non-ORB applications and resources.
- Interoperability between the object transaction service model and the X/Open Distributed Transaction Processing (DTP) model.
- Access to existing (non-object) programs and resource managers by objects.
- Access to objects by existing programs and resource managers.
- Coordination by a single transaction service of the activities of both object and non-object resource managers.
- The network case: A single transaction, distributed between an object and non-object system, each of which has its own Transaction Service.

The Transaction Service accommodates this requirement for implementations where interoperability with X/Open DTP-compliant transactional applications is necessary.

Network Interoperability. Customers require the ability to interoperate between systems offered by multiple vendors:

- Single transaction service, single ORB - It must be possible for a single transaction service to interoperate with itself using a single ORB.
- Multiple transaction services, single ORB - It must be possible for one transaction service to interoperate with a cooperating transaction service using a single ORB.
- Single transaction service, multiple ORBs - It must be possible for a single transaction service to interoperate with itself using different ORBs.

- Multiple transaction services, multiple ORBs - It must be possible for one transaction service to interoperate with a cooperating transaction service using different ORBs.

The Transaction Service specifies all required interactions between cooperating Transaction Service implementations necessary to support a single ORB. The Transaction Service depends on ORB interoperability as defined in the *Common Object Request Broker: Architecture and Specification* (CORBA specification) to provide cooperating Transaction Services across different ORBs.

Flexible transaction propagation control. Both client and object implementations can control transaction propagation:

- A client controls whether or not its transaction is propagated with an operation.
- A client can invoke operations on objects with transactional behavior and objects without transactional behavior within the scope of a single transaction.
- An object can specify transactional behavior for its interfaces.

The Transaction Service supports both implicit (system-managed) propagation and explicit (application-managed) propagation. With implicit propagation, transactional behavior is not specified in the operation's signature. With explicit propagation, applications define their own mechanisms for sharing a common transaction.

Support for TP Monitors. Customers need object technology to build mission-critical applications. These applications are deployed on commercial transaction processing systems where a TP Monitor provides both efficient scheduling and the sharing of resources by a large number of users. It must be possible to implement the Transaction Service in a TP monitor environment. This includes the ability to execute:

- multiple transactions concurrently
- clients, servers, and transaction services in separate processes.

The Transaction Service is usable in a TP Monitor environment.

1.2.5.2 Design Requirements

The Transaction Service supports the following design requirements:

Exploitation of OO Technology. This specification permits a wide variety of ORB and Transaction Service implementations and uses objects to enable ORB-based, secure implementations. The Transaction Service provides the programmer with easy to use interfaces that hide some of the complexity inherent in general-use specifications. Meaningful user applications can be constructed using interfaces that are as simple or simpler than their procedural equivalents.

Low Implementation Cost. The Transaction Service specification considers cost from the perspective of three users of the service - clients, ORB implementers, and Transaction Service providers.

- For clients, it allows a range of implementations that are compliant with the proposed architecture. Many ORB implementations will exist in client workstations, which have no requirement to understand transactions within themselves, but will find it highly desirable to interoperate with server platforms that implement transactions.
- The specification provides for minimal impact to the ORB. Where feasible, function is assigned to an object service implementation to permit the ORB to continue to provide high performance object access when transactions are not used.
- Since this Transaction Service will be supported by existing (procedural) transaction managers, the specification allows implementations that reuse existing procedural Transaction Managers.

Portability. The Transaction Service specification provides for portability of applications. It also defines an interface between the ORB and the Transaction Service that enables individual Transaction Service implementations to be ported between different ORB implementations.

Avoidance of OMG IDL interface variants. The Transaction Service allows a single interface to be supported by both transactional and non-transactional implementations. This approach avoids a potential "combinatorial explosion" of interface variants that differ only in their transactional characteristics. For example, the existing Object Service interfaces can support transactional behavior without change.

Support for both single-threaded and multi-threaded implementations. The Transaction Service defines a flexible model that supports a variety of programming styles. For example, a client with an active transaction can make requests for the same transaction on multiple threads. Similarly, an object can support multiple transactions in parallel by using multiple threads.

A wide spectrum of implementation choices. The Transaction Service allows implementations to choose the degree of checking provided to guarantee legal behavior of its users. This permits both robust implementations that provide strong assurances for transaction integrity and lightweight implementations where such checks are not warranted.

1.2.5.3 Performance Requirements

The Transaction Service is expected to be implemented on a wide range of hardware and software platforms ranging from desktop computers to massively parallel servers and in networks ranging in size from a single LAN to worldwide networks. To meet this wide range of requirements, consideration must be given to algorithms that scale, efficient communications, and the number and size of accesses to permanent storage. Much of this is implementation, and therefore not visible to the user of the service. Nevertheless, the expected performance of the Transaction Service was compared to its procedural equivalent, the X/Open DTP model in the following areas:

- The number of network messages required.
- The number of disk accesses required.
- The amount of data logged.

The objective of the specification was to achieve parity with the X/Open model for equivalent function, where technically feasible.

1.3 Service Architecture

Figure 1-2 illustrates the major components and interfaces defined by the Transaction Service. The transaction originator is an arbitrary program that begins a transaction. The recoverable server implements an object with recoverable state that is invoked within the scope of the transaction, either directly by the transaction originator or indirectly through one or more transactional objects.

The transaction originator creates a transaction using a *TransactionFactory*; a *Control* is returned that provides access to a *Terminator* and a *Coordinator*. The transaction originator uses the *Terminator* to commit or rollback the transaction. The *Coordinator* is made available to recoverable servers, either explicitly or implicitly (by implicitly propagating a transaction context with a request). A recoverable server registers a *Resource* with the *Coordinator*. The *Resource* implements the two-phase commit protocol which is driven by the Transaction Service. A recoverable server may register a *Synchronization* with the *Coordinator*. The *Synchronization* implements a dependent object protocol driven by the Transaction Service. A recoverable server can also register a specialized resource called a *SubtransactionAwareResource* to track the completion of subtransactions. A *Resource* uses a *RecoveryCoordinator* in certain failure cases to determine the outcome of the transaction and to coordinate the recovery process with the Transaction Service.

To simplify coding, most applications use the *Current* pseudo object, which provides access to an implicit per-thread transaction context.

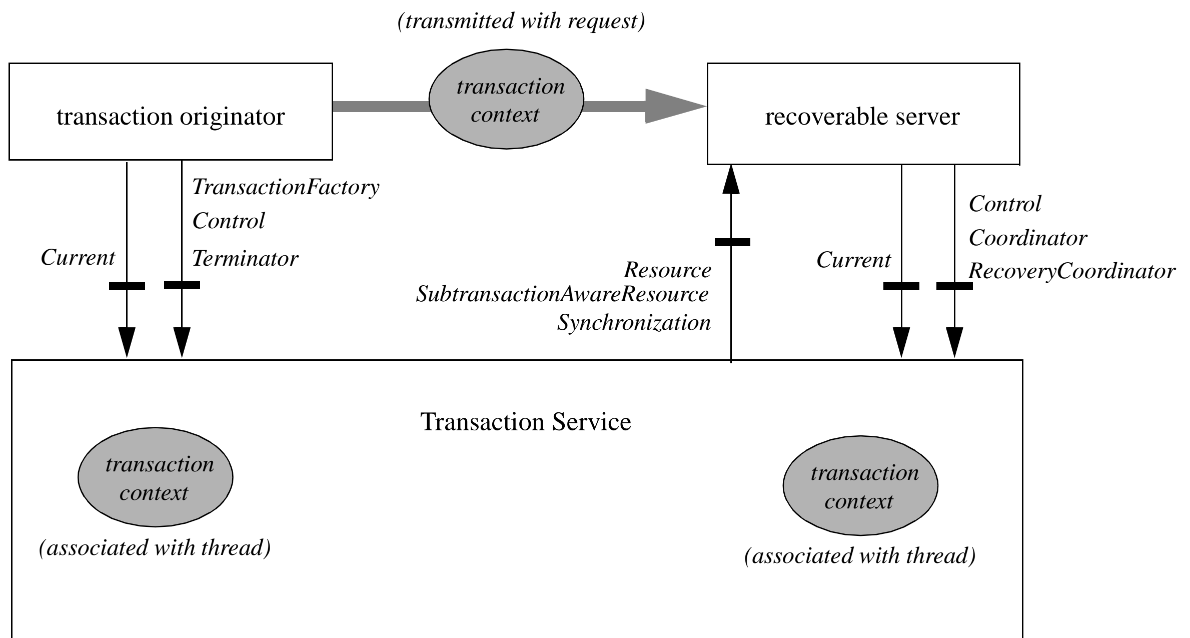

Figure 1-2 Major Components and Interfaces of the Transaction Service

1.3.1 Typical Usage

A typical transaction originator uses the *Current* object to begin a transaction, which becomes associated with the transaction originator's thread.

The transaction originator then issues requests. Some of these requests involve transactional objects. When a request is issued to a transactional object, the transaction context associated with the invoking thread is automatically propagated to the thread executing the method of the target object. No explicit operation parameter or context declaration is required to transmit the transaction context. Propagation of the transaction context can extend to multiple levels if a transactional object issues a request to a transactional object.

Using the *Current* object, the transactional object can unilaterally rollback the transaction and can inquire about the current state of the transaction. Using the *Current* object, the transactional object also can obtain a *Coordinator* for the current transaction. Using the *Coordinator*, a transactional object can determine the relationship between two transactions, to implement isolation among multiple transactions.

Some transactional objects are also recoverable objects. A recoverable object has persistent data that must be managed as part of the transaction. A recoverable object uses the *Coordinator* to register a *Resource* object as a participant in the transaction. The resource represents the recoverable object's participation in the transaction; each resource is implicitly associated with a single transaction. The *Coordinator* uses the resource to perform the two-phase commit protocol on the recoverable object's data.

After the computations involved in the transaction have been completed, the transaction originator uses the *Current* object to request that the changes be committed. The Transaction Service commits the transaction using a two-phase commit protocol wherein a series of requests are issued to the registered resources.

1.3.2 Transaction Context

The transaction context associated with a thread is either null (indicating that the thread has no associated transaction) or it refers to a specific transaction. It is permitted for multiple threads to be associated with the same transaction at the same time.

When a thread in an object server is used by an object adapter to perform a request on a transactional object, the object adapter initializes the transaction context associated with that thread by effectively copying the transaction context of the thread that issued the request. An implementation of the Transaction Service may restrict the capabilities of the new transaction context. For example, an implementation of the Transaction Service might not permit the object server thread to request commitment of the transaction.

The object adapter initializes the transaction context of the request handler only if a Transaction Service provided *IOP::ServiceContext* is present in the request message. Otherwise, the initial transaction context of the thread is empty because there is no transaction.

When a thread retrieves the response to a deferred synchronous request, an exception may be raised if the thread is no longer associated with the transaction that it was associated with when the deferred synchronous request was issued. See the *Common Object Request Broker: Architecture and Specification*, *Dynamic Invocation Interface* chapter for a definition of the “WrongTransaction Exception” and for its usage in **get_response** and **get_next_response** respectively.

When nested transactions are used, the transaction context remembers the stack of nested transactions started within a particular execution environment (e.g., process) so that when a subtransaction ends, the transaction context of the thread is restored to the context in effect when the subtransaction was begun. When the context is transferred between execution environments, the received context refers only to one particular transaction, not a stack of transactions.

1.3.3 Context Management

The Transaction Service supports management and propagation of transaction context using objects provided by the Transaction Service. Using this approach, the transaction originator issues a request to a *TransactionFactory* to begin a new top-level transaction. The factory returns a *Control* object specific to the new transaction that allows an application to terminate the transaction or to become a participant in the transaction (by registering a *Resource*). An application can propagate a transaction context by passing the *Control* as an explicit request parameter.

The *Control* does not directly support management of the transaction. Instead, it supports operations that return two other objects, a *Terminator* and a *Coordinator*. The *Terminator* is used to commit or rollback the transaction. The *Coordinator* is used to enable transactional objects to participate in the transaction. These two objects can be propagated independently, allowing finer granularity control over propagation.

An application can also use the *Current* object operations **get_control**, **suspend**, and **resume** to obtain or change the implicit transaction context associated with its thread.

When nested transactions are used, a *Control* can include a stack of nested transactions begun in the same execution environment. When a *Control* is transferred between execution environments, the received *Control* refers only to one particular transaction, not a stack of transactions.

1.3.4 Datatypes

The **CosTransactions** module defines the following datatypes:

```
enum Status {
    StatusActive,
    StatusMarkedRollback,
    StatusPrepared,
    StatusCommitted,
```

```
        StatusRolledBack,
        StatusUnknown,
        StatusNoTransaction,
        StatusPreparing,
        StatusCommitting,
        StatusRollingBack
};

enum Vote {
        VoteCommit,
        VoteRollback,
        VoteReadOnly
};
// Old definitions are retained for backward compatibility. //
// TransactionPolicyValue definitions are deprecated and replaced with new//
// InvocationPolicy and OTSPolicy definitions which are defined below //
// Old definitions are retained for backward compatibility. //

typedef unsigned short TransactionPolicyValue;

const TransactionPolicyValue    Allows_shared = 0;
const TransactionPolicyValue    Allows_none = 1;
const TransactionPolicyValue    Requires_shared = 2;
const TransactionPolicyValue    Allows_unshared = 3;
const TransactionPolicyValue    Allows_either = 4;
const TransactionPolicyValue    Requires_unshared = 5;
const TransactionPolicyValue    Requires_either = 6;

typedef unsigned short InvocationPolicyValue;

const InvocationPolicyValue EITHER = 0;
const InvocationPolicyValue SHARED = 1;
const InvocationPolicyValue UNSHARED =2;

typedef unsigned short OTSPolicyValue;

const OTSPolicyValue REQUIRES = 1;
const OTSPolicyValue FORBIDS =2;
const OTSPolicyValue ADAPTS =3;

typedef unsigned short NonTxTargetPolicyValue;

const NonTxTargetPolicyValue PREVENT = 0;
const NonTxTargetPolicyValue PERMIT = 1;
```

The **CosTSInteroperation** module defines the following datatypes:

```
const IOP::ComponentId TAG_TRANSACTION_POLICY= 26;

struct TransactionPolicyComponent {
        CosTransactions::TransactionPolicyValue tpv;
```

```
};

const IOP::ComponentId TAG_OTS_POLICY= 31;

const IOP::ComponentId TAG_INV_POLICY= 32;
```

1.3.5 Structures

The **CosTransactions** module defines the following structures:

```
struct otid_t {
    long formatID; /*format identifier. 0 is OSI TP */
    long bqual_length;
    sequence <octet> tid;
};
struct TransIdentity {
    Coordinator coord;
    Terminator term;
    otid_t otid;
};
struct PropagationContext {
    unsigned long timeout;
    TransIdentity current;
    sequence <TransIdentity> parents;
    any implementation_specific_data;
};
```

1.3.6 Exceptions

1.3.6.1 Standard Exceptions

The standard exceptions TRANSACTION_REQUIRED, TRANSACTION_ROLLEDBACK, INVALID_TRANSACTION, TRANSACTION_UNAVAILABLE, and TRANSACTION_MODE are related to the Transaction Service. These exceptions are defined in the *Common Object Request Broker: Architecture and Specification* (*ORB Interface* chapter, *Standard Exception Definitions* section).

1.3.6.2 Heuristic Exceptions

A heuristic decision is a unilateral decision made by one or more participants in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the Transaction Service. A participant can only make a heuristic decision once the two-phase-commit protocol has begun, in particular it cannot make such a decision if it receives a rollback without a previous prepare. Heuristic decisions are normally made only in unusual circumstances, such as

communication failures, that prevent normal processing. When a heuristic decision is taken, there is a risk that the decision will differ from the consensus outcome, resulting in a loss of data integrity.

The **CosTransactions** module defines the following exceptions for reporting incorrect heuristic decisions or the possibility of incorrect heuristic decisions:

```
exception HeuristicRollback {};
exception HeuristicCommit {};
exception HeuristicMixed {};
exception HeuristicHazard {};
```

HeuristicRollback Exception

The **commit** operation on *Resource* raises the HeuristicRollback exception to report that a heuristic decision was made and that all relevant updates have been rolled back.

HeuristicCommit Exception

The **rollback** operation on *Resource* raises the HeuristicCommit exception to report that a heuristic decision was made and that all relevant updates have been committed.

HeuristicMixed Exception

A request raises the HeuristicMixed exception to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

HeuristicHazard Exception

A request raises the HeuristicHazard exception to report that a heuristic decision may have been made, the disposition of all relevant updates is not known, and for those updates whose disposition is known, either all have been committed or all have been rolled back. (In other words, the HeuristicMixed exception takes priority over the HeuristicHazard exception.)

TRANSACTION_UNAVAILABLE Exception

The **CosTransactions** module adds the TRANSACTION_UNAVAILABLE exception that can be raised by the ORB when it cannot process a transaction service context because its connection to the Transaction Service has been abnormally terminated. This exception is defined in the *Common Object Request Broker: Architecture and Specification*, *IDL Syntax and Semantics* chapter.

TRANSACTION_MODE Exception

The **CosTransactions** module adds the TRANSACTION_MODE exception that can be raised by the ORB when it detects a mismatch between the **TransactionPolic**y in the IOR and the current transaction mode. This exception is defined in the *Common Object Request Broker: Architecture and Specification*, *IDL Syntax and Semantics* chapter.

1.3.6.3 Other Exceptions

The **CosTransactions** module defines the following additional exceptions:

```
exception SubtransactionsUnavailable {};
exception NotSubtransaction {};
exception Inactive {};
exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
exception SynchronizationUnavailable {};
```

These exceptions are described in Chapter 2 along with the operations that raise them.

Transaction Service Interfaces 2

Note – All text in black is from the Transaction Service, v1.2 (formal/01-11-03). Text in blue is from the Components specification.

Contents

This chapter contains the following sections.

Section Title	Page
"Introduction"	2-2
"Current Interface"	2-2
"TransactionFactory Interface"	2-5
"Control Interface"	2-6
"Terminator Interface"	2-7
"Coordinator Interface"	2-8
"Recovery Coordinator Interface"	2-13
"Resource Interface"	2-14
"Synchronization Interface"	2-16
"Subtransaction Aware Resource Interface"	2-17
"TransactionalObject Interface"	2-18
"Policy Interfaces"	2-18
"The User's View"	2-26
"The Implementers' View"	2-40

2.1 Introduction

The interfaces defined by the Transaction Service reside in the **CosTransactions** module (see Appendix A - OMG IDL for the **CosTransactions** module IDL). The interfaces for the Transaction Service are as follows:

- Current
- TransactionFactory
- Terminator
- Coordinator
- RecoveryCoordinator
- Resource
- Synchronization
- Subtransaction Aware Resource

No operations are defined in these interfaces for destroying objects. No application actions are required to destroy objects that support the Transaction Service because the Transaction Service destroys its own objects when they are no longer needed.

2.2 Current Interface

The **Current** interface defines operations that allow a client of the Transaction Service to explicitly manage the association between threads and transactions. The **Current** interface also defines operations that simplify the use of the Transaction Service for most applications. These operations can be used to begin and end transactions and to obtain information about the current transaction.

The **Current** interface is a locality-constrained interface whose behavior depends upon and may alter the transaction context associated with the invoking thread. It is obtained by using a resolve initial references (“TransactionCurrent”) operation on the **CORBA::ORB** interface. **Current** supports the following operations:

```
local interface Current : CORBA::Current {
    void begin()
        raises(SubtransactionsUnavailable);
    void commit(in boolean report_heuristics)
        raises(
            NoTransaction,
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback()
        raises(NoTransaction);
    void rollback_only()
        raises(NoTransaction);

    Status get_status();
    string get_transaction_name();
    void set_timeout(in unsigned long seconds)
```

```
    unsigned long get_timeout();

    Control get_control();
    Control suspend();
    void resume(in Control which)
        raises(InvalidControl);
};
```

Note – In order to pass the transaction from one thread to another, a program should not use the Current object. It should pass the Control object to the other thread.

2.2.1 begin

A new transaction is created. The transaction context of the client thread is modified so that the thread is associated with the new transaction. If the client thread is currently associated with a transaction, the new transaction is a subtransaction of that transaction. Otherwise, the new transaction is a top-level transaction.

The SubtransactionsUnavailable exception is raised if the client thread already has an associated transaction and the Transaction Service implementation does not support nested transactions.

2.2.2 commit

If there is no transaction associated with the client thread, the NoTransaction exception is raised. If the client thread does not have permission to commit the transaction, the standard exception NO_PERMISSION is raised. (The **commit** operation may be restricted to the transaction originator in some implementations.)

Otherwise, the transaction associated with the client thread is completed. The effect of this request is equivalent to performing the **commit** operation on the corresponding **Terminator** object (see Section 2.5, "Terminator Interface," on page 2-7 and Section 1.3.6, "Exceptions," on page 1-16 for a description of the exceptions that may be raised).

The client thread transaction context is modified as follows: If the transaction was begun by a thread (invoking **begin**) in the same execution environment, then the thread's transaction context is restored to its state prior to the **begin** request. Otherwise, the thread's transaction context is set to null.

2.2.3 rollback

If there is no transaction associated with the client thread, the NoTransaction exception is raised. If the client thread does not have permission to rollback the transaction, the standard exception NO_PERMISSION is raised. (The **rollback** operation may be restricted to the transaction originator in some implementations; however, the **rollback_only** operation, described below, is available to all transaction participants.)

Otherwise, the transaction associated with the client thread is rolled back. The effect of this request is equivalent to performing the **rollback** operation on the corresponding **Terminator** object (see Section 2.5, “Terminator Interface,” on page 2-7).

The client thread transaction context is modified as follows: If the transaction was begun by a thread (invoking **begin**) in the same execution environment, then the thread’s transaction context is restored to its state prior to the **begin** request. Otherwise, the thread’s transaction context is set to null.

2.2.4 rollback_only

If there is no transaction associated with the client thread, the NoTransaction exception is raised. Otherwise, the transaction associated with the client thread is modified so that the only possible outcome is to rollback the transaction. The effect of this request is equivalent to performing the **rollback_only** operation on the corresponding **Coordinator** object (see Section 2.6, “Coordinator Interface,” on page 2-8).

2.2.5 get_status

If there is no transaction associated with the client thread, the **StatusNoTransaction** value is returned. Otherwise, this operation returns the status of the transaction associated with the client thread. The effect of this request is equivalent to performing the **get_status** operation on the corresponding **Coordinator** object (see Section 2.6, “Coordinator Interface,” on page 2-8).

2.2.6 get_transaction_name

If there is no transaction associated with the client thread, an empty string is returned. Otherwise, this operation returns a printable string describing the transaction. The returned string is intended to support debugging. The effect of this request is equivalent to performing the **get_transaction_name** operation on the corresponding **Coordinator** object (see Section 2.6, “Coordinator Interface,” on page 2-8).

2.2.7 set_timeout

This operation modifies a state variable associated with the target object that affects the time-out period in number of seconds associated with top-level transactions created by subsequent invocations of the begin operation. If the parameter has a non-zero value n, then top-level transactions created by subsequent invocations of begin will be subject to being rolled back if they do not complete before n seconds after their creation. Nested transactions are not subject to such time-outs and will only be rolled back automatically if their enclosing top-level transaction is rolled back. If the parameter is zero, then no application specified time-out is established.

2.2.8 get_timeout

This operation returns the state variable associated with the target object that affects the time-out period in number of seconds associated with top-level transactions created by invocations of the **begin** operation, or 0 if no application specific time-out has been established.

2.2.9 get_control

If the client thread is not associated with a transaction, a null object reference is returned. Otherwise, a **Control** object is returned that represents the transaction context currently associated with the client thread. This object can be given to the **resume** operation to re-establish this context in the same thread or a different thread. The scope within which this object is valid is implementation dependent; at a minimum, it must be usable by the client thread. This operation is not dependent on the state of the transaction; in particular, it does not raise the TRANSACTION_ROLLEDBACK exception.

2.2.10 suspend

If the client thread is not associated with a transaction, a null object reference is returned. Otherwise, an object is returned that represents the transaction context currently associated with the client thread. This object can be given to the **resume** operation to re-establish this context in the same thread or a different thread. The scope within which this object is valid is implementation dependent; at a minimum, it must be usable by the client thread. In addition, the client thread becomes associated with no transaction. This operation is not dependent on the state of the transaction; in particular, it does not raise the TRANSACTION_ROLLEDBACK exception.

2.2.11 resume

If the parameter is a null object reference, the client thread becomes associated with no transaction. Otherwise, if the parameter is valid in the current execution environment, the client thread becomes associated with that transaction (in place of any previous transaction). Otherwise, the InvalidControl exception is raised. See Section 2.4, “Control Interface,” on page 2-6 for a discussion of restrictions on the scope of a **Control**. This operation is not dependent on the state of the transaction; in particular, it does not raise the TRANSACTION_ROLLEDBACK exception.

2.3 TransactionFactory Interface

The **TransactionFactory** interface is provided to allow the transaction originator to begin a transaction. This interface defines two operations, **create** and **recreate**, which create a new representation of a top-level transaction.

```
interface TransactionFactory {
    Control create(in unsigned long time_out);
    Control recreate(in PropagationContext ctx);
};
```

2.3.1 create

A new top-level transaction is created and a **Control** object is returned. The **Control** object can be used to manage or to control participation in the new transaction. An implementation of the Transaction Service may restrict the ability for the **Control** object to be transmitted to or used in other execution environments. At a minimum, it can be used by the client thread.

If the parameter has a nonzero value **n**, then the new transaction will be subject to being rolled back if it does not complete before **n** seconds have elapsed. If the parameter is zero, then no application specified time-out is established.

2.3.2 recreate

A new representation is created for an existing transaction defined by the **PropagationContext** and a **Control** object is returned. The **Control** object can be used to manage or to control participation in the transaction. An implementation of the Transaction Service, which supports interposition (see Section 2.14.2, "ORB/TS Implementation Considerations," on page 2-51) uses **recreate** to create a new representation of the transaction being imported, subordinate to the representation in **ctx**. The **recreate** operation can also be used to import a transaction that originated outside of the Transaction Service.

2.4 Control Interface

The **Control** interface allows a program to explicitly manage or propagate a transaction context. An object supporting the **Control** interface is implicitly associated with one specific transaction.

```
interface Control {
    Terminator get_terminator()
        raises(Unavailable);
    Coordinator get_coordinator()
        raises(Unavailable);
};
```

The **Control** interface defines two operations, **get_terminator** and **get_coordinator**. The **get_terminator** operation returns a **Terminator** object, which supports operations to end the transaction. The **get_coordinator** operation returns a **Coordinator** object, which supports operations needed by resources to participate in the transaction. The two objects support operations that are typically performed by different parties. Providing two objects allow each set of operations to be made available only to the parties that require those operations.

A **Control** object for a transaction is obtained using the operations defined by the **TransactionFactory** interface or the **create_subtransaction** operation defined by the **Coordinator** interface. It is possible to obtain a **Control** object for the current transaction (associated with a thread) using the **get_control** or **suspend** operations defined by the **Current** interface (see Section 2.2, "Current Interface," on page 2-2). (These two operations return a null object reference if there is no current transaction.)

An implementation of the Transaction Service may restrict the ability for the **Control** object to be transmitted to or used in other execution environments; at a minimum, it can be used within a single thread.

2.4.1 get_terminator

An object is returned that supports the **Terminator** interface. The object can be used to rollback or commit the transaction associated with the **Control**. The Unavailable exception may be raised if the **Control** cannot provide the requested object. An implementation of the Transaction Service may restrict the ability for the **Terminator** object to be transmitted to or used in other execution environments. At a minimum, it can be used within the client thread.

2.4.2 get_coordinator

An object is returned that supports the **Coordinator** interface. The object can be used to register resources for the transaction associated with the **Control**. The Unavailable exception may be raised if the *Control* cannot provide the requested object. An implementation of the Transaction Service may restrict the ability for the **Coordinator** object to be transmitted to or used in other execution environments. At a minimum, it can be used within the client thread.

2.5 Terminator Interface

The **Terminator** interface supports operations to commit or rollback a transaction. Typically, these operations are used by the transaction originator.

```
interface Terminator {
    void commit(in boolean report_heuristics)
        raises(
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback();
};
```

An implementation of the Transaction Service may restrict the scope in which a **Terminator** can be used. At a minimum, it can be used within a single thread.

2.5.1 commit

If the transaction has not been marked rollback only, and all of the participants in the transaction agree to commit, the transaction is committed and the operation terminates normally. Otherwise, the transaction is rolled back (as described below) and the TRANSACTION_ROLLEDBACK standard exception is raised.

The **report_heurisitcs** parameter allows the application to control how long it will block after issuing a commit. If the **report_heuristics** parameter is true, the call will block until phase 2 of the commit protocol is complete and all heuristic outcomes are known. The Transaction Service will report inconsistent or possibly inconsistent outcomes using the HeuristicMixed and HeuristicHazard exceptions (defined in Section 1.3.6, "Exceptions," on page 1-16). If the parameter is false, a conforming Transaction Service must not raise these exceptions. Transaction Service implementations may make use of this fact to block only to the end of phase 1 when the outcome is known, but heuristics are still possible. Heuristics that do occur may be reported to some management interface that is more suited to taking recovery action than the application.

The **commit** operation may rollback the transaction if there are subtransactions of the transaction that have not themselves been committed or rolled back, or if there are existing or potential activities associated with the transaction that have not completed. The nature and extent of such error checking is implementation-dependent.

When a top-level transaction is committed, all changes to recoverable objects made in the scope of this transaction are made permanent and visible to other transactions or clients. When a subtransaction is committed, the changes are made visible to other related transactions as appropriate to the degree of isolation enforced by the resources.

2.5.2 rollback

The transaction is rolled back. When a transaction is rolled back, all changes to recoverable objects made in the scope of this transaction (including changes made by descendant transactions) are rolled back. All resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the resources.

2.6 Coordinator Interface

The **Coordinator** interface provides operations that are used by participants in a transaction. These participants are typically either recoverable objects or agents of recoverable objects, such as subordinate coordinators. Each object supporting the **Coordinator** interface is implicitly associated with a single transaction.

```
interface Coordinator {

    Status get_status();
    Status get_parent_status();
    Status get_top_level_status();
```

```
        boolean is_same_transaction(in Coordinator tc);
        boolean is_related_transaction(in Coordinator tc);
        boolean is_ancestor_transaction(in Coordinator tc);
        boolean is_descendant_transaction(in Coordinator tc);
        boolean is_top_level_transaction();

        unsigned long hash_transaction();
        unsigned long hash_top_level_tran();

        RecoveryCoordinator register_resource(in Resource r)
            raises(Inactive);

        void register_synchronization (in Synchronization sync)
            raises(Inactive, SynchronizationUnavailable);

    };

        void register_subtran_aware(in SubtransactionAwareResource r)
            raises(Inactive, NotSubtransaction);

        void rollback_only()
            raises(Inactive);

        string get_transaction_name();

        Control create_subtransaction()
            raises(SubtransactionsUnavailable, Inactive);

        PropagationContext get_txcontext ()
            raises(Unavailable);
    };
```

An implementation of the Transaction Service may restrict the scope in which a **Coordinator** can be used. At a minimum, it can be used within a single thread.

2.6.1 *get_status*

This operation returns the status of the transaction associated with the target object:

- **StatusActive** - A transaction is associated with the target object and it is in the active state. An implementation returns this status after a transaction has been started and prior to a coordinator issuing any prepares unless it has been marked for rollback.
- **StatusMarkedRollback** - A transaction is associated with the target object and has been marked for rollback, perhaps as the result of a **rollback_only** operation.
- **StatusPrepared** - A transaction is associated with the target object and has been prepared (i.e., all subordinates have responded **VoteCommit**). The target object may be waiting for a superior's instructions as to how to proceed.

- **StatusCommitted** - A transaction is associated with the target object and it has completed commitment. It is likely that heuristics exists; otherwise, the transaction would have been destroyed and **StatusNoTransaction** returned.
- **StatusRolledBack** - A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exists; otherwise, the transaction would have been destroyed and **StatusNoTransaction** returned.
- **StatusUnknown** - A transaction is associated with the target object, but the Transaction Service cannot determine its current status. This is a transient condition, and a subsequent invocation will ultimately return a different status.
- **StatusNoTransaction** - No transaction is currently associated with the target object. This will occur after a transaction has completed.
- **StatusPreparing** - A transaction is associated with the target object and it is the process of preparing. An implementation returns this status if it has started preparing, but has not yet completed the process, probably because it is waiting for responses to prepare from one or more resources.
- **StatusCommitting** - A transaction is associated with the target object and is in the process of committing. An implementation returns this status if it has decided to commit, but has not yet completed the process, probably because it is waiting for responses from one or more resources.
- **StatusRollingBack** - A transaction is associated with the target object and it is in the process of rolling back. An implementation returns this status if it has decided to rollback, but has not yet completed the process, probably because it is waiting for responses from one or more resources.

2.6.2 *get_parent_status*

If the transaction associated with the target object is a top-level transaction, then this operation is equivalent to the **get_status** operation. Otherwise, this operation returns the status of the parent of the transaction associated with the target object.

2.6.3 *get_top_level_status*

This operation returns the status of the top-level ancestor of the transaction associated with the target object. If the transaction is a top-level transaction, then this operation is equivalent to the **get_status** operation.

2.6.4 *is_same_transaction*

This operation returns true if, and only if, the target object and the parameter object both refer to the same transaction.

2.6.5 *is_ancestor_transaction*

This operation returns true if, and only if, the transaction associated with the target object is an ancestor of the transaction associated with the parameter object. A transaction T1 is an ancestor of a transaction T2 if, and only if, T1 is the same as T2 or T1 is an ancestor of the parent of T2.

2.6.6 *is_descendant_transaction*

This operation returns true if, and only if, the transaction associated with the target object is a descendant of the transaction associated with the parameter object. A transaction T1 is a descendant of a transaction T2 if, and only if, T2 is an ancestor of T1 (see above).

2.6.7 *is_related_transaction*

This operation returns true if, and only if, the transaction associated with the target object is related to the transaction associated with the parameter object. A transaction T1 is related to a transaction T2 if, and only if, there exists a transaction T3 such that T3 is an ancestor of T1 and T3 is an ancestor of T2.

2.6.8 *is_top_level_transaction*

This operation returns true if, and only if, the transaction associated with the target object is a top-level transaction. A transaction is a top-level transaction if it has no parent.

2.6.9 *hash_transaction*

This operation returns a hash code for the transaction associated with the target object. Each transaction has a single hash code. Hash codes for transactions should be uniformly distributed.

2.6.10 *hash_top_level_tran*

This operation returns the hash code for the top-level ancestor of the transaction associated with the target object. This operation is equivalent to the **hash_transaction** operation when the transaction associated with the target object is a top-level transaction.

2.6.11 *register_resource*

This operation registers the specified resource as a participant in the transaction associated with the target object. When the transaction is terminated, the resource will receive requests to commit or rollback the updates performed as part of the transaction. These requests are described in the description of the **Resource** interface. The

Inactive exception is raised if the transaction has already been prepared. The standard exception TRANSACTION_ROLLEDBACK may be raised if the transaction has been marked rollback only.

If the resource is a subtransaction aware resource (it supports the **SubtransactionAwareResource** interface) and the transaction associated with the target object is a subtransaction, then this operation registers the specified resource with the subtransaction and indirectly with the top-level transaction when the subtransaction's ancestors have committed.

If the resource is not a subtransaction aware resource and the transaction associated with the target object is a subtransaction, then the resource is registered as a participant of this subtransaction. It is registered with the parent of this subtransaction only if and when this subtransaction is committed. Otherwise (the transaction is a top-level transaction), the resource is registered as a participant in this transaction.

This operation returns a **RecoveryCoordinator** that can be used by this resource during recovery.

2.6.12 register_synchronization

This operation registers the specified **Synchronization** object such that it will be notified to perform necessary processing prior to prepare being driven to resources registered with this **Coordinator**. These requests are described in the description of the **Synchronization** interface. The Inactive exception is raised if the transaction has already been prepared. The SynchronizationUnavailable exception is raised if the **Coordinator** does not support synchronization. The standard exception TRANSACTION_ROLLEDBACK may be raised if the transaction has been marked rollback only.

A synchronization cannot be registered with a subtransaction. A call to **register_synchronization** on a subtransaction always raises SynchronizationUnavailable.

2.6.13 register_subtran_aware

This operation registers the specified subtransaction aware resource such that it will be notified when the subtransaction has committed or rolled back. These requests are described in the description of the **SubtransactionAwareResource** interface.

Note that this operation registers the specified resource only with the subtransaction. This operation cannot be used to register the resource as a participant in the transaction.

The NotSubtransaction exception is raised if the current transaction is not a subtransaction. The Inactive exception is raised if the subtransaction (or any ancestor) is terminating, or has already been terminated. The standard exception TRANSACTION_ROLLEDBACK may be raised if the subtransaction (or any ancestor) has been marked rollback only.

2.6.14 rollback_only

The transaction associated with the target object is modified so that the only possible outcome is to rollback the transaction. The Inactive exception is raised if the transaction has already been prepared.

2.6.15 get_transaction_name

This operation returns a printable string describing the transaction associated with the target object. The returned string is intended to support debugging.

2.6.16 create_subtransaction

A new subtransaction is created whose parent is the transaction associated with the target object. The Inactive exception is raised if the target transaction is terminating, or has already been terminated. An implementation of the Transaction Service is not required to support nested transactions. If nested transactions are not supported, the exception SubtransactionsUnavailable is raised.

The **create_subtransaction** operation returns a **Control** object, which enables the subtransaction to be terminated and allows recoverable objects to participate in the subtransaction. An implementation of the Transaction Service may restrict the ability for the **Control** object to be transmitted to or used in other execution environments.

2.6.17 get_txcontext

The **get_txcontext** operation returns a **PropagationContext** object, which is used by one Transaction Service domain to export the current transaction to a new Transaction Service domain. An implementation of the Transaction Service may also use the **PropagationContext** to assist in the implementation of the **is_same_transaction** operation when the input **Coordinator** has been generated by a different Transaction Service implementation.

The Unavailable exception is raised if the Transaction Service implementation chooses to restrict the availability of the **PropagationContext**.

2.7 Recovery Coordinator Interface

A recoverable object uses a **RecoveryCoordinator** to drive the recovery process in certain situations. The object reference for an object supporting the **RecoveryCoordinator** interface, as returned by the **register_resource** operation, is implicitly associated with a single resource registration request and may only be used by that resource.

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

2.7.1 replay_completion

This operation can be invoked at any time after the associated resource has been prepared. The **Resource** must be passed as the parameter. Performing this operation provides a hint to the **Coordinator** that the **commit** or **rollback** operations have not been performed on the resource. This hint may be required in certain failure cases. This non-blocking operation returns the current status of the transaction. The NotPrepared exception is raised if the resource has not been prepared.

2.8 Resource Interface

The Transaction Service uses a two-phase commitment protocol to complete a top-level transaction with each registered resource. The **Resource** interface defines the operations invoked by the transaction service on each resource. Each object supporting the **Resource** interface is implicitly associated with a single top-level transaction. Note that in the case of failure, the completion sequence will continue after the failure is repaired. A resource should be prepared to receive duplicate requests for the **commit** or **rollback** operation and to respond consistently.

```
interface Resource {
    Vote prepare()
        raises(
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback()
        raises(
            HeuristicCommit,
            HeuristicMixed,
            HeuristicHazard
        );
    void commit()
        raises(
            NotPrepared,
            HeuristicRollback,
            HeuristicMixed,
            HeuristicHazard
        );
    void commit_one_phase()
        raises(
            HeuristicHazard
        );
    void forget();
};
```

2.8.1 prepare

This operation is invoked to begin the two-phase commit protocol on the resource. The resource can respond in several ways, represented by the **Vote** result.

If no persistent data associated with the resource has been modified by the transaction, the resource can return **VoteReadOnly**. After receiving this response, the Transaction Service is not required to perform any additional operations on this resource. Furthermore, the resource can forget all knowledge of the transaction.

If the resource is able to write (or has already written) all the data needed to commit the transaction to stable storage, as well as an indication that it has prepared the transaction, it can return **VoteCommit**. After receiving this response, the Transaction Service is required to eventually perform either the **commit** or the **rollback** operation on this object. To support recovery, the resource should store the **RecoveryCoordinator** object reference in stable storage.

The resource can return **VoteRollback** under any circumstances, including not having any knowledge about the transaction (which might happen after a crash). If this response is returned, the transaction must be rolled back. Furthermore, the Transaction Service is not required to perform any additional operations on this resource. After returning this response, the resource can forget all knowledge of the transaction.

The resource reports inconsistent outcomes using the HeuristicMixed and HeuristicHazard exceptions (described in Section 1.3.6, "Exceptions," on page 1-16). Heuristic outcomes occur when a resource acts as a sub-coordinator and at least one of its resources takes a heuristic decision after a **VoteCommit** return. If a heuristic outcome exception is raised, the resource must remember this outcome until the **forget** operation is performed so that it can return the same outcome in case **commit** or **rollback** is performed.

2.8.2 rollback

If necessary, the resource should rollback all changes made as part of the transaction. If the resource has forgotten the transaction, it should do nothing.

The heuristic outcome exceptions (described in Section 1.3.6, "Exceptions," on page 1-16) are used to report heuristic decisions related to the resource. The resource may raise these exceptions only if the **prepare** operation has been performed previously. If a heuristic outcome exception is raised, the resource must remember this outcome until the **forget** operation is performed so that it can return the same outcome in case **rollback** is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

2.8.3 commit

If necessary, the resource should commit all changes made as part of the transaction. If the resource has forgotten the transaction, it should do nothing.

The heuristic outcome exceptions (described in Section 1.3.6, "Exceptions," on page 1-16) are used to report heuristic decisions related to the resource. If a heuristic outcome exception is raised, the resource must remember this outcome until the **forget** operation is performed so that it can return the same outcome in case **commit** is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

The NotPrepared exception is raised if the **commit** operation is performed without first performing the **prepare** operation.

2.8.4 commit_one_phase

If possible, the resource should commit all changes made as part of the transaction. If it cannot, it should raise the TRANSACTION_ROLLEDBACK standard exception.

If a failure occurs during **commit_one_phase**, it must be retried when the failure is repaired. Since their can only be a single resource, the HeuristicHazard exception is used to report heuristic decisions related to that resource. If a heuristic exception is raised, the resource must remember this outcome until the **forget** operation is performed so that it can return the same outcome in case **commit_one_phase** is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

2.8.5 forget

This operation is performed only if the resource raised a heuristic outcome exception to **rollback**, **commit**, **commit_one_phase**, or **prepare**. Once the coordinator has determined that the heuristic situation has been addressed, it should issue **forget** on the resource. The resource can forget all knowledge of the transaction.

2.9 Synchronization Interface

The Transaction Service provides a synchronization protocol, which enables an object with transient state data that relies on an X/Open XA conformant Resource Manager for ensuring that data is made persistent to be notified before the start of the two-phase commitment protocol, and after its completion. If the transaction is instructed to roll back rather than be committed, the object will only be notified after rollback completes. An object with transient state data that relies on a **Resource** object for ensuring that data is made persistent can also make use of this protocol, provided that both objects are registered with the same **Coordinator**. Each object supporting the **Synchronization** interface is implicitly associated with a single top-level transaction.

```
// TransactionalObject has been deprecated //
// and replaced by the use of the OTSPolicy component //
// Synchronization will use the OTSPolicy of ADAPTS //
// Inheritance from TransactionalObject is for backward compatability //

interface Synchronization : TransactionalObject {
    void before_completion();
    void after_completion(in Status s);
};
```

2.9.1 before_completion

This operation is invoked prior to the start of the two-phase commit protocol within the coordinator the **Synchronization** has registered with. This operation will therefore be invoked prior to **prepare** being issued to **Resource** objects or X/Open Resource Managers registered with that same coordinator. The **Synchronization** object must ensure that any state data it has that needs to be made persistent is made available to the resource.

Only standard exceptions may be raised. Unless there is a defined recovery procedure for the exception raised, the transaction should be marked rollback only.

2.9.2 after_completion

Regardless of how the transaction was originally instructed to terminate, this operation is invoked after all commit or rollback responses have been received by this coordinator. The current status of the transaction (as determined by a **get_status** on the Coordinator) is provided as input.

Only standard exceptions may be raised and they have no effect on the outcome of the transaction termination.

2.10 Subtransaction Aware Resource Interface

Recoverable objects that implement nested transaction behavior may support a specialization of the **Resource** interface called the **SubtransactionAwareResource** interface. A recoverable object can be notified of the completion of a subtransaction by registering a specialized resource object that offers the **SubtransactionAwareResource** interface with the Transaction Service. This registration is done by using the **register_resource** or the **register_subtran_aware** operation of the current **Coordinator** object. A recoverable object generally uses the **register_resource** operation to register a resource that will participate in the completion of the top-level transaction and the **register_subtran_aware** operation to be notified of the completion of a subtransaction.

Certain recoverable objects may want a finer control over the registration in the completion of a subtransaction. These recoverable objects will use the **register_resource** operation to ensure participation in the completion of the top-level transaction and they will use the **register_subtran_aware** operation to be notified of the completion of a particular subtransaction. For example, a recoverable object can use the **register_subtran_aware** operation to establish a “committed with respect to” relationship between transactions; that is, the recoverable object wants to be informed when a particular subtransaction is committed and then perform certain operations on the transactions that depend on that transaction’s completion. This technique could be used to implement lock inheritance, for example.

The Transaction Service uses the **SubtransactionAwareResource** interface on each **Resource** object registered with a subtransaction. Each object supporting this interface is implicitly associated with a single subtransaction.

```
interface SubtransactionAwareResource : Resource {
    void commit_subtransaction(in Coordinator parent);
    void rollback_subtransaction();
};
```

2.10.1 commit_subtransaction

This operation is invoked only if the resource has been registered with a subtransaction and the subtransaction has been committed. The **Resource** object is provided with a **Coordinator** that represents the parent transaction. This operation may raise a standard exception such as TRANSACTION_ROLLEDBACK.

Note that the results of a committed subtransaction are relative to the completion of its ancestor transactions, that is, these results can be undone if any ancestor transaction is rolled back.

2.10.2 rollback_subtransaction

This operation is invoked only if the resource has been registered with a subtransaction and notifies the resource that the subtransaction has rolled back.

2.11 TransactionalObject Interface

The **TransactionalObject** interface is a remnant of previous versions of this specification and is no longer used. It is retained here only for backward compatibility with OTS 1.0 and OTS 1.1.

```
interface TransactionalObject{
};
```

2.12 Policy Interfaces

The Transaction Service utilizes POA policies to define characteristics related to transactions. These policies are encoded in the IOR as tag components and exported to the client when an object reference is created. This enables validation that a particular object is capable of supporting the transaction characteristics expected by the client.

Background

The introduction of asynchronous messaging (AMI) into CORBA requires a new form of transaction model to be supported. The current CORBA model, the *shared transaction model*, provides an end to end transaction shared by the client and the server. This model cannot be supported by asynchronous messaging. Instead, a new model, which uses a store and forward transport between the client and server, is introduced. In this new model, the communication between client and server is broken into separate requests, separated by a reliable transmission between routers. When

transaction are used, this model uses multiple shared transactions, each executed to completion before the next one begins. This transaction model is called the *unshared transaction model.*

Design Rationale

Introducing the unshared transaction model into CORBA requires enhancements to the current method for specifying transactional behavior, which currently defines only the shared transaction model. The different models of transactional behaviors are more properly implementation properties, suggesting that they not be declared in interfaces. Instead they are specified by the server using POA policies and made available to the client via IOR profile components.

In OTS 1.0 and OTS 1.1, an object declared its ability to support a shared transaction by inheriting from an empty interface called **TransactionalObject**. This mechanism had weak transaction semantics, since it was also used by the infrastructure to control transaction propagation. Such an object always received a shared transaction if one was active, but did not receive one when there was no active transaction. This behavior is more accurately described as **allowing** a shared transaction, since it provided no guarantee to the client as to what the object might do if it did or did not receive a shared transaction. This weak semantic is not carried forward as an explicit policy. OTS 1.0 and OTS 1.1 did not provide a mechanism to **require** a shared transaction at invocation time. This behavior produces the following two by two matrix of possible choices for shared transaction support.

Table 2-1 Shared Transaction Behaviors

Transaction	None	Shared
Requires	no inheritance from **TransactionalObject**	cannot be specified with OTS 1.1
Allows	no inheritance from **TransactionalObject**	inheritance from **TransactionalObject**

- **cell (1,1)** - the object requires ‘no transaction’
- **cell (1,2)** - the object requires a shared transaction
- **cell (2,1)** - the object allows ‘no transaction’
- **cell (2,2)** - the object allows a shared transaction

OTSPolicy

Although the use of **TransactionalObject** is maintained for backward compatability, explicit transactional behaviors are now encoded using **OTSPolicy** values, which are independent of the transaction propagation rules used by the infrastructure. These policies and their OTS 1.1 equivalents are defined as shown in Table 2-2

Table 2-2 New Shared Transaction Behaviors

OTSPolicy	Policy Value	OTS 1.1 Equivalent
Reserved [1]	0	inheritance from **TransactionalObject**
REQUIRES	1	No equivalent
FORBIDS	2	no inheritance from **TransactionalObject** [2]
ADAPTS [3]	3	No equivalent

[1] - The ALLOWS semantics associated with inheritance from **TransactionalObject** cannot be coded as an explicit **OTSPolicy** value in OTS 1.2.

[2] - FORBIDS is more restrictive than the absence of inheritance from **TransactionalObject** since it may raise the INVALID_TRANSACTION exception.

[3] - ADAPTS provides a stronger client-side guarantee than inheritance from **TransactionalObject**.

- REQUIRES - The behavior of the target object depends on the existence of a current transaction. If the invocation does not have a current transaction, a TRANSACTION_REQUIRED exception will be raised.
- FORBIDS - The behavior of the target object depends on the absence of a current transaction. If the invocation does have a current transaction, an INVALID_TRANSACTION exception will be raised.
- ADAPTS - The behavior of the target object will be adjusted to take advantage of a current transaction, if one exists. If not, it will exhibit a different behavior (i.e., the target object is sensitive to the presence or absence of a current transaction).

OTSPolicy values are encoded in the **TAG_OTS_POLICY** component of the IOR and will always be present when IORs are created by OTS-aware ORBs at the OTS 1.2 level or above. If an **OTSPolicy** is not present in the IOR, the client may assume that it was created by an OTS-unaware ORB or an OTS-aware ORB at the OTS 1.1 level or below. To distinguish the two, it is necessary to determine whether the target object inherits from **TransactionalObject** and use the mapping defined in Table 2-3.

Table 2-3 Mapping empty TAG_OTS_POLICY components to OTSPolicy values

Inheritance from TransactionalObject	OTSPolicy Value
NO	FORBIDS[1]
YES	ADAPTS[2]

[1] - The mapping applies to the policy checking rules only. FORBIDS has stronger transactional semantics than the absence of inheritance from **TransactionalObject**.

[2] - The mapping applies to the policy checking rules only. ADAPTS has stronger transactional semantics than the use of inheritance from **TransactionalObject**.

An **OTSPolicy** object cannot be associated with a POA on an OTS-unaware ORB so those POAs create IORs with no **TAG_OTS_POLICY** component.

InvocationPolicy

With the introduction of messaging, the unshared transaction model is used when the request is made via a router. The **InvocationPolicy** specifies which form of invocation is supported by the target object. The **InvocationPolicy** is defined in Table 2-4.

Table 2-4 InvocationPolicy Behaviors

InvocationPolicy	Policy Value
EITHER	0
SHARED	1
UNSHARED	2

- EITHER - The behavior of the target is not affected by the mode of client invocation. Both direct invocations (synchronous) and invocations using routers (asynchronous) are supported.
- SHARED - all invocations which do not involve a routing element (i.e., the client ORB directly invokes the target object with no intermediate routers). This includes:
  - synchronous stub based invocations,
  - synchronous or deferred synchronous invocations using Dynamic Invocation Interface (DII),
  - Asynchronous Method Invocations (AMI) with an effective **RoutingPolicy** of **ROUTE_NONE**.
- UNSHARED - all invocations that involve a routing element. This includes Asynchronous Method Invocations (AMI) with an effective **RoutingPolicy** of **ROUTE_FORWARD** or **ROUTE_STORE_AND_FORWARD**.

The **InvocationPolicy** component is significant only when transactions are used with CORBA messaging.

InvocationPolicy values are encoded in the **TAG_INV_POLICY** component of the IOR. If an **InvocationPolicy** is not present in the IOR, it is interpreted as if the **TAG_INV_POLICY** was present with a value of EITHER.

Interactions between InvocationPolicy and OTSPolicy

Although **InvocationPolicy** and **OTSPolicy** are distinct policies, not all combinations are valid. The valid choices are shown in Table 2-5.

Table 2-5 InvocationPolicy and OTSPolicy combinations

InvocationPolicy/ OTSPolicy	EITHER	SHARED	UNSHARED
REQUIRES	ok Requires_either	ok Requires_shared	ok Requires_unshared

Table 2-5 InvocationPolicy and OTSPolicy combinations

InvocationPolicy/ OTSPolicy	EITHER	SHARED	UNSHARED
FORBIDS	invalid	ok Allows_none	invalid
ADAPTS	invalid	ok Allows_shared	invalid

Transactional target objects that accept invocations via routers must support shared transactions, since the routers use the shared transaction model to reliably forward the request to the next router or the eventual target object.

Invalid policy combinations are detected when the POA is created (see Section 2.12.1, "Creating Transactional Object References," on page 2-23).

NonTxTargetPolicy Policy

The **NonTxTargetPolicy** Policy is used to control the ability for clients to make requests on non-transactional objects while in the scope of an active transaction. A non-transactional object has an IOR that either contains a **TAG_OTS_POLICY** component with a value of FORBIDS or does not contain a **TAG_OTS_POLICY** component at all. The **NonTxTargetPolicy** policy is an ORB-policy that is set by the client application using the **ORB::create_policy** interface. Once set, the policy is used to control whether requests on non-transactional targets will raise the INVALID_TRANSACTION exception (PREVENT) or will be permitted to proceed normally (PERMIT). The default **NonTxTargetPolicy** is **PERMIT**.

Policy Interface Definitions

The new policy interfaces are defined in the **CosTransactions** module. These interfaces are defined by the following OMG IDL:

```
module CosTransactions {

    // TransactionPolicyType is deprecated and replaced //
    // by InvocationPolicyType and OTSPolicyType //
    // It is retained for backward compatibility. //

    typedef unsigned short TransactionPolicyValue;

    const CORBA::PolicyType    TransactionPolicyType = 46;

    interface TransactionPolicy : CORBA::Policy {
        readonly attribute TransactionPolicyValue tpv;
    };

    const CORBA::PolicyType    INVOCATION_POLICY_TYPE = 55;
```

```
    typedef unsigned short InvocationPolicyValue;

    interface InvocationPolicy : CORBA::Policy {
        readonly attribute InvocationPolicyValue ipv;
    };

    const CORBA::PolicyType OTS_POLICY_TYPE = 56;

    typedef unsigned short OTSPolicyValue;

    interface OTSPolicy : CORBA::Policy {
        readonly attribute OTSPolicyValue tpv;
    };

    const CORBA::PolicyType NON_TX_TARGET_POLICY_TYPE = 57;

    typedef unsigned short NonTxTargetPolicyValue;

    interface NonTxTargetPolicy : CORBA::Policy {
        readonly attribute NonTxTargetPolicyValue tpv;
    };
};
```

2.12.1 Creating Transactional Object References

Object references are created as defined by the POA. An **OTSPolicy** object is created by invoking **ORB::create_policy** with a **PolicyType** of **OTSPolicyType** and a value of type **OTSPolicyValue**. An **InvocationPolicy** may also be associated with a POA using the same mechanism. When either or both of these policies are associated with a POA, the POA will create object references with either or both policies encoded as tagged components in the IOR:

- **OTSPolicy** objects can only be used with POAs that support an OTS-aware ORB at the OTS 1.2 level or above.
- **InvocationPolicy** objects can only be used with POAs that support an OTS-aware ORB at the OTS 1.2 level or above.

If a POA is not created with either policy object on an OTS-aware ORB at the OTS 1.2 level or higher, that POA is created as if an **OTSPolicy** object were present with its value set to FORBIDS.

If a POA is created on an OTS-unaware ORB, that POA creates object references that do not include either tag component.

Transaction-unaware POAs

A transaction-unaware POA is any POA created on an OTS-unaware ORB. A transaction-unaware POA will never create a **TAG_OTS_POLICY** component in any IORs it creates. Transaction-unaware POAs cannot be created on an OTS-aware ORB with an associated OTS 1.2 or higher implementation, however it is possible to create a POA that does not support transactions on an OTS-aware ORB (see the next section).

Transaction-aware POAs

A transaction-aware POA is any POA which is created on an OTS-aware ORB with an associated OTS 1.2 or higher implementation. A transaction-aware POA will include tag components in IORs it creates for **OTSPolicy** values and optionally **InvocationPolicy** values.

- Transaction-aware POAs can only be created in a server, which has an OTS 1.2 or higher implementation associated with its ORB (i.e., an OTS-aware ORB).
- If an application attempts to create a POA with an **OTSPolicy** object in a server that does not have an associated OTS (i.e., an OTS-unaware ORB), the InvalidPolicy exception is raised.
- A POA that does not support transactions is created in an OTS-aware ORB with an **OTSPolicy** object with a FORBIDS policy value and is still called a transaction-aware POA.
- Transaction-aware POAs must have at least an **OTSPolicy** object associated with them. If an **OTSPolicy** is not provided explicitly, an **OTSPolicy** object is created implicitly with a value of FORBIDS.
- Transaction-aware POAs may (but need not) have **InvocationPolicy** objects associated with them.
- An attempt to create a transaction-aware POA with conflicting **OTSPolicy** and **InvocationPolicy** values (as defined in Table 2-5 on page 2-21) will raise the InvalidPolicy exception.

Table 2-6 summarizes the relationship between POA creation and IOR components on both OTS-unaware and OTS-aware ORBs.

Table 2-6 POA creation and IOR components

create_POA	**OTS-unaware ORB**	**OTS-aware ORB**		
POA Policies	**Result**	**Result**	**TAG_INV_POLICY**	**TAG_OTS_POLICY**
Neither	ok	ok	NO	YES (with FORBIDS)
InvocationPolicy SHARED	raise InvalidPolicy	ok	YES	YES (with FORBIDS)
InvocationPolicy EITHER or UNSHARED	raise InvalidPolicy	raise InvalidPolicy	-	-

Table 2-6 POA creation and IOR components

create_POA	OTS-unaware ORB	OTS-aware ORB		
POA Policies	**Result**	**Result**	**TAG_INV_POLICY**	**TAG_OTS_POLICY**
OTSPolicy	raise InvalidPolicy	ok	NO	YES
Both with valid combinations	raise InvalidPolicy	ok	YES	YES
Both with invalid combinations	raise InvalidPolicy	raise InvalidPolicy	-	-

Impact of Transactions on the POA

When there is a current transaction established, the POA's **Servant** location function is performed within the scope of that transaction. The POA is responsible for making sure that all invocations on a Servant Locator, which can result in reading or writing persistent storage (**pre_invoke** and **post_invoke**) execute within the scope of the current transaction. Activators are not invoked as part of the transaction. The following behaviors must be made explicit:

- A Servant Locator cannot send the operation reply to the client until **post_invoke** has completed successfully.
- Certain failures in these operation calls take precedence over sending replies (e.g., **TRANSACTION_ROLLEDBACK**) and must be raised back to the client.
- **ServantActivator** and **AdapterActivator** invocations are not within the scope of the transaction. An Activator implementation must start its own transaction if its actions are to take place within a transaction.

Appearance of Policy Components in IORs

The **OTSPolicyValue** and **InvocationPolicyValue** are encoded as CDR encapsulations in the **TAG_OTS_POLICY** and **TAG_INVOCATION_POLICY** **TaggedComponents** of the IOR. The tags of these **TaggedComponents** are defined in the following IDL:

```
// The TAG_TRANSACTION_POLICY component is deprecated and //
// replaced by InvocationPolicy and OTSPolicy components //
// It is retained for backward compatibility only. //

module CosTSInteroperation {

    const IOP::ComponentId TAG_TRANSACTION_POLICY=26:

    struct TransactionPolicyComponent {
        CosTransactions::TransactionPolicyValue tpv;
    };
```

```
const IOP::ComponentId TAG_OTS_POLICY= 31;

const IOP::ComponentId TAG_INV_POLICY= 32;

};
```

2.13 The User's View

The audience for this section is object and client implementers; it describes application use of the Transaction Service functions.

2.13.1 Application Programming Models

A client application program may use direct or indirect context management to manage a transaction.

- With indirect context management, an application uses the **Current** object provided by the Transaction Service, to associate the transaction context with the application thread of control.
- In direct context management, an application manipulates the **Control** object and the other objects associated with the transaction.

Propagation is the act of associating a client's transaction context with operations on a target object. An object may require transactions to be either explicitly or implicitly propagated on its operations.

Implicit propagation means that requests are implicitly associated with the client's transaction; they share the client's transaction context. It is transmitted implicitly to the objects, without direct client intervention. Implicit propagation depends on indirect context management, since it propagates the transaction context associated with the **Current** object. ***Explicit propagation*** means that an application propagates a transaction context by passing objects defined by the Transaction Service as explicit parameters.

An object that supports implicit propagation would not typically expect to receive any Transaction Service object as an explicit parameter.

A client may use one or both forms of context management, and may communicate with objects that use either method of transaction propagation.

This results in four ways in which client applications may communicate with transactional objects. They are described below.

2.13.1.1 Direct Context Management: Explicit Propagation

The client application directly accesses the **Control** object, and the other objects that describe the state of the transaction. To propagate the transaction to an object, the client must include the appropriate Transaction Service object as an explicit parameter of an operation.

2.13.1.2 Indirect Context Management: Implicit Propagation

The client application uses operations on the **Current** object to create and control its transactions. When it issues requests on transactional objects, the transaction context associated with the current thread is implicitly propagated to the object.

2.13.1.3 Indirect Context Management: Explicit Propagation

For an implicit model application to use explicit propagation, it can get access to the **Control** using the **get_control** operation on **Current**. It can then use a Transaction Service object as an explicit parameter to a transactional object. This is explicit propagation.

2.13.1.4 Direct Context Management: Implicit Propagation

A client that accesses the Transaction Service objects directly can use the **resume** operation on **Current** to set the implicit transaction context associated with its thread. This allows the client to invoke operations of an object that requires implicit propagation of the transaction context.

2.13.2 Interfaces

Table 2-7 Use of Transaction Service Functionality

Function	Used by	Context management Direct	Indirect[1]
Create a transaction	Transaction originator	TransactionFactory::create Control::get_terminator Control::get_coordinator	begin,set_timeout
Terminate a transaction	Transaction originator—*implicit* All—*explicit*	Terminator::commit Terminator::rollback	commit rollback
Rollback a transaction	Server	Terminator::rollback_only	rollback_only
Control propagation of transaction to a server	Server	Declaration of method parameter	begin resume
Control by client of transaction propagation to a server	All	Request parameters	get_control suspend resume
Become a participant in a transaction	Recoverable Server	Coordinator::register_resource	Not applicable
Miscellaneous	All	Coordinator::get_status Coordinator::get_transaction_name Coordinator::is_same_transaction Coordinator::hash_transaction	get_status get_transaction_name Not applicable Not applicable

1. All Indirect context management operations are on the *Current* object interface

Note – For clarity, subtransaction operations are not shown.

2.13.3 Checked Transaction Behavior

Some Transaction Service implementations will enforce checked behavior for the transactions they support, to provide an extra level of transaction integrity. The purpose of the checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. A checked Transaction Service guarantees that commit will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests.

There are many possible implementations of checking in a Transaction Service. One provides equivalent function to that provided by the request/response inter-process communication models defined by X/Open.

The X/Open Transaction Service model of checking is particularly important because it is widely implemented. It describes the transaction integrity guarantees provided by many existing transaction systems. These transaction systems will provide the same level of transaction integrity for object-based applications by providing a Transaction Service interface that implements the X/Open checks.

2.13.4 X/Open Checked Transactions

In X/Open, completion of the processing of a request means that the object has completed execution of its method and replied to the request.

The level of transaction integrity provided by a Transaction Service implementing the X/Open model of checking provides equivalent function to that provided by the XATMI and TxRPC interfaces defined by X/Open for transactional applications. X/Open DTP Transaction Managers are examples of transaction management functions that implement checked transaction behavior.

This implementation of checked behavior depends on implicit transaction propagation. When implicit propagation is used, the objects involved in a transaction at any given time may be represented as a tree, the request tree for the transaction. The beginner of the transaction is the root of the tree. Requests add nodes to the tree, replies remove the replying node from the tree. Synchronous requests, or the checks described below for deferred synchronous requests, ensure that the tree collapses to a single node before commit is issued.

If a transaction uses explicit propagation, the Transaction Service cannot know which objects are or will be involved in the transaction; that is, a request tree cannot be constructed or assured. Therefore, the use of explicit propagation is not permitted by a Transaction Service implementation that enforces X/Open-style checked behavior.

Applications that use synchronous requests implicitly exhibit checked behavior. For applications that use deferred synchronous requests, in a transaction where all clients and objects are in the domain of a checking Transaction Service, the Transaction Service can enforce this property by applying a reply check and a commit check.

The Transaction Service must also apply a resume check to ensure that the transaction is only resumed by application programs in the correct part of the request tree.

2.13.4.1 Reply Check

Before allowing an object to reply to a transactional request, a check is made to ensure that the object has received replies to all its deferred synchronous requests that propagated the transaction in the original request. If this condition is not met, an exception is raised and the transaction is marked as rollback-only, that is, it cannot be successfully committed.

A Transaction Service may check that a reply is issued within the context of the transaction associated with the request.

2.13.4.2 Commit Check

Before allowing commit to proceed, a check is made to ensure that:

- The commit request for the transaction is being issued from the same execution environment that created the transaction.
- The client issuing commit has received replies to all the deferred synchronous requests it made that caused the propagation of the transaction.

2.13.4.3 Resume Check

Before allowing a client or object to associate a transaction context with its thread of control, a check is made to ensure that this transaction context was previously associated with the execution environment of the thread. This would be true if the thread either created the transaction or received it in a transactional operation.

2.13.5 Implementing a Transactional Client: Heuristic Completions

The **commit** operation takes the boolean **report_heuristics** as input. If the **report_heuristics** argument is *false*, **commit** can complete as soon as the root coordinator has made its decision to commit or rollback the transaction. The application is not required to wait for the coordinator to complete the commit protocol by informing all the participants of the outcome of the transaction. This can significantly reduce the elapsed time for the commit operation, especially where participant **Resource** objects are located on remote network nodes. However, no heuristic conditions can be reported to the application in this case.

Using the **report_heuristics** option guarantees that the **commit** operation will not complete until the coordinator has completed the commit protocol with all resources involved in the transaction. This guarantees that the application will be informed of any non-atomic outcomes of the transaction via the HeuristicMixed or HeuristicHazard exceptions, but increases the application-perceived elapsed time for the **commit** operation.

2.13.6 Implementing a Recoverable Server

A Recoverable Server includes at least one recoverable object and one **Resource** object. The responsibilities of each of these objects are explained in the following sections.

2.13.6.1 Recoverable Object

The responsibilities of the recoverable object are to implement the object's operations, and to register a **Resource** object with the **Coordinator** so commitment of the recoverable object's resources, including any necessary recovery, can be completed.

The **Resource** object identifies the involvement of the recoverable object in a particular transaction. This means a **Resource** object may only be registered in one transaction at a time. A different **Resource** object must be registered for each transaction in which a recoverable object is concurrently involved.

A recoverable object may receive multiple requests within the scope of a single transaction. It only needs to register its involvement in the transaction once. The **is_same_transaction** operation allows the recoverable object to determine if the transaction associated with the request is one in which the recoverable object is already registered.

The **hash_transaction** operations allow the recoverable object to reduce the number of transaction comparisons it has to make. All coordinators for the same transaction return the same hash code. The **is_same_transaction** operation need only be done on coordinators that have the same hash code as the coordinator of the current request.

2.13.6.2 Resource Object

The responsibilities of a **Resource** object are to participate in the completion of the transaction, to update the Recoverable Server's resources in accordance with the transaction outcome, and ensure termination of the transaction, including across failures. The protocols that the **Resource** object must follow are described in Section 2.14.1, "Transaction Service Protocols," on page 2-40.

2.13.6.3 Reliable Servers

A Reliable Server is a special case of a Recoverable Server. A Reliable Server can use the same interface as a Recoverable Server to ensure application integrity for objects that do not have recoverable state. In the case of a Reliable Server, the recoverable object can register a **Resource** object that replies **VoteReadOnly** to prepare if its integrity constraints are satisfied (e.g., all debits have a corresponding credit), or replies **VoteRollback** if there is a problem. This approach allows the server to apply integrity constraints that apply to the transaction as a whole, rather than to individual requests to the server.

2.13.7 Application Portability

This section considers application portability across the broadest range of Transaction Service implementations.

2.13.7.1 Flat Transactions

There is one optional function of the Transaction Service, support for nested transactions. For an application to be portable across all implementations of the Transaction Service, it should be designed to use the flat transaction model. The Transaction Service specification treats flat transactions as top-level nested transactions.

2.13.7.2 X/Open Checked Transactions

Transaction Service implementations may implement checked or unchecked behavior. The transaction integrity checks implemented by a Transaction Service need not be the same as those defined by X/Open. However, many existing transaction management systems have implemented the X/Open model of interprocess communication, and will implement a checked Transaction Service that provides the same guarantee of transaction integrity.

Applications written to conform to the transaction integrity constraints of X/Open will be portable across all implementations of an X/Open checked Transaction Service, as well as all Transaction Service implementations that support unchecked behavior.

2.13.8 Distributed Transactions

The Transaction Service can be implemented by multiple components located across a network. The different components can be based on the same or on different implementations of the Transaction Service.

A single transaction can involve clients and objects supported by more than one instance of the Transaction Service. The number of Transaction Service instances involved in the transaction is not visible to the application implementer. There is no change in the function provided.

2.13.9 Applications Using Both Checked and Unchecked Services

A single transaction can include objects supported by both checked and unchecked Transaction Service implementations. Checked transaction behavior cannot be applied to the transaction as a whole.

It is possible to provide useful, limited forms of checked behavior for those subsets of the transaction's resources in the domain of a checked Transaction Service.

- First, a transactional or recoverable object, whose resources are managed by a checked Transaction Service, may be accessed by unchecked clients. The checked Transaction Service can ensure, by registering itself in the transaction, that the transaction will not commit before all the integrity constraints associated with the request have been satisfied.
- Second, an application whose resources are managed by a checked Transaction Service may act as a client of unchecked objects, and preserve its checked semantics.

2.13.10 Examples

Note – All the examples are written in pseudo code based on C++. In particular they do not include implicit parameters such as the **ORB::Environment**, which should appear in all requests. Also, they do not handle the exceptions that might be returned with each request.

2.13.10.1 A Transaction Originator: Indirect and Implicit

In the code fragments below, a transaction originator uses indirect context management and implicit transaction propagation; **txn_crt** is an example of an object supporting the **Current** interface. The client uses the **begin** operation to start the transaction which becomes implicitly associated with the originator's thread of control.

```
...
txn_crt.begin();
// should test the exceptions that might be raised
...
// the client issues requests, some of which involve
// transactional objects;
BankAccount1->makeDeposit(deposit);
...
```

The program **commit**s the transaction associated with the client thread. The **report_heuristics** argument is set to *false* so no report will be made by the Transaction Service about possible heuristic decisions.

```
....
txn_crt.commit(false);
...
```

2.13.10.2 Transaction Originator: Direct and Explicit

In the following example, a transaction originator uses direct context management and explicit transaction propagation. The client uses a factory object supporting the **CosTransactions::TransactionFactory** interface to create a new transaction and uses the returned **Control** object to retrieve the **Terminator** and **Coordinator** objects.

```
...
CosTransactions::Control c;
CosTransactions::Terminator t;
CosTransactions::Coordinator co;

c = TFactory->create(0);
t = c->get_terminator();
...
```

The client issues requests, some of which involve transactional objects, in this case explicit propagation of the context is used. The **Control** object reference is passed as an explicit parameter of the request; it is declared in the OMG IDL of the interface.

```
...
transactional_object->do_operation(arg, c);
```

The transaction originator uses the **Terminator** object to commit the transaction; the **report_heuristics** argument is set to *false*: so no report will be made by the Transaction Service about possible heuristic decisions.

```
...
t->commit(false);
```

2.13.10.3 *Example of a Recoverable Server*

BankAccount1 is an object with internal resources. It inherits from the **Resource** interfaces:

```
interface BankAccount1:
    CosTransactions::Resource
{
...
  void makeDeposit (in float amt);
...
};


class BankAccount1
{
public:
...
void makeDeposit(float amt);
...
}
```

Upon entering, the context of the transaction is implicitly associated with the object's thread. The pseudo object supporting the **Current** interface is used to retrieve the **Coordinator** object associated with the transaction.

```
void makeDeposit (float amt)
{
CosTransactions::Control c;
CosTransactions::Coordinator co;

c = txn_crt.get_control();
co = c->get_coordinator();
...
```

Before registering the **Resource**, the object must check whether it has already been registered for the same transaction. This is done using the **hash_transaction** and **is_same_transaction** operations on the current **Coordinator** to compare a list of saved coordinators representing currently active transactions. In this example, the object registers itself as a **Resource**. This requires the object to durably record its registration before issuing **register_resource** to handle potential failures and imposes the restriction that the object may only be involved in one transaction at a time.

If more parallelism is required, separate **Resource** objects can be registered for each transaction the object is involved in.

```
RecoveryCoordinator r;
r = co->register_resource (this);

// performs some transactional activity locally
balance = balance + f;
num_transactions++;
...
// end of transactional operation
};
```

2.13.10.4 Example of a Transactional Object

BankAccount2 is an object with external resources.

```
interface BankAccount2 {
...
  void makeDeposit(in float amt);
...
};


class BankAccount2
{
public:
...
void makeDeposit(float amt);
...
}
```

Upon entering, the context of the transaction is implicitly associated with the object's thread. The **makeDeposit** operation performs some transactional requests on external, recoverable servers. The objects **res1** and **res2** are recoverable objects. The current transaction context is implicitly propagated to these objects.

```
void makeDeposit(float amt)
{
     balance = res1->get_balance(amt);
     balance = balance + amt;
     res1->set_balance(balance);

     res2->increment_num_transactions();
} // end of transactional operation
```

2.13.11 Model Interoperability

The Transaction Service supports interoperability between Transaction Service applications using implicit context propagation and procedural applications using the X/Open DTP model. A single transaction management component may act as both the Transaction Service and an X/Open Transaction Manager.

Interoperability is provided in two ways:

- Importing transactions from the X/Open domain to the Transaction Service domain.
- Exporting transactions from the Transaction Service domain to the X/Open domain.

2.13.11.1 Importing Transactions

X/Open applications can access transactional objects. This means that an existing application, written to use X/Open interfaces, can be extended to invoke transactional operations. This causes the X/Open transaction to be imported into the domain of the Transaction Service.

The X/Open application may be a client or a server.

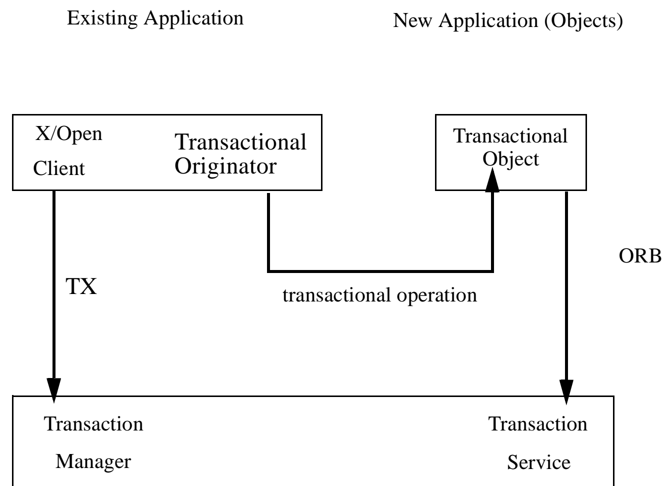

Figure 2-1 X/Open Client

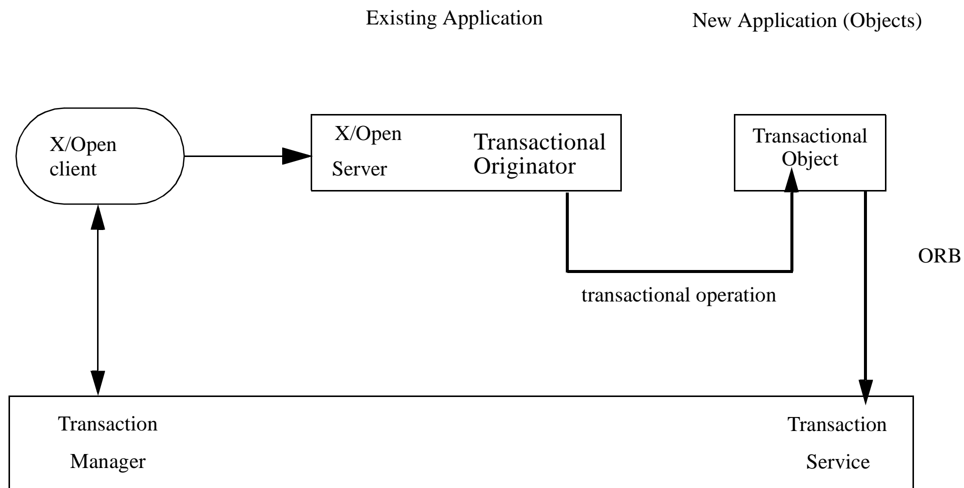

Figure 2-2 X/Open Server

2.13.11.2 Exporting Transactions

Transactional objects can use X/Open communications and resource manager interfaces, and include the resources managed by these components in a transaction managed by the Transaction Service. This causes the Transaction Service transaction to be exported into the domain of the X/Open transaction manager.

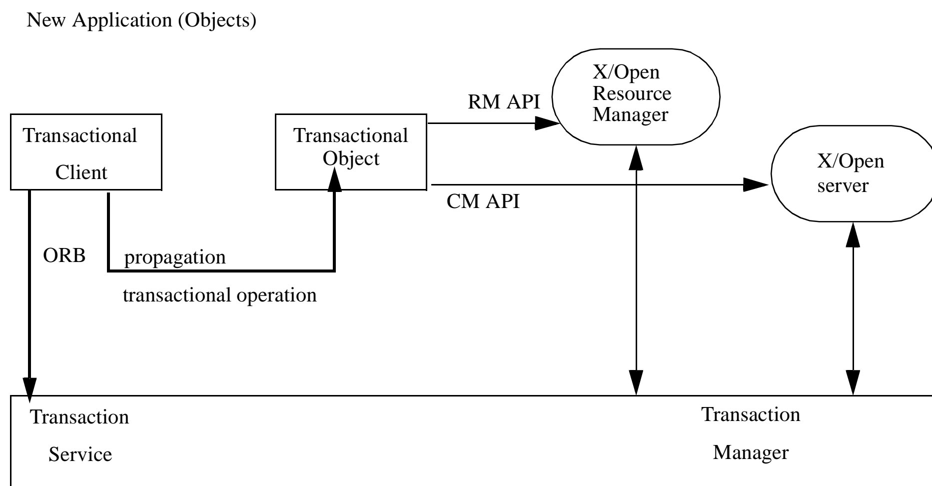

Figure 2-3 Sample Transaction Managed by the Transaction Service

2.13.11.3 Programming Rules

Model interoperability results in application programs that use both X/Open and Transaction Service interfaces.

A transaction originator may use the X/Open TX interface or the Transaction Service interfaces to create and terminate a transaction. Only one style may be used in one originator.

A single application may inherit a transaction with an application request either by using the X/Open server interfaces, or by being a transactional object.

Within a single transaction, an application program can be a client of both X/Open resource manager interfaces and transactional object interfaces.

An X/Open client or server may invoke operations of transactional objects. The X/Open transaction is imported into the Transaction Service domain using the **recreate** operation on **TransactionFactory**.

A transactional object with a **Current** object that associates a transaction context with a thread of control, can call X/Open Resource Managers. How requests to the X/Open Resource managers become associated with the transaction context of the **Current** object is implementation dependent.

2.13.12 Failure Models

The Transaction Service provides atomic outcomes for transactions in the presence of application, system, or communication failures. This section describes the behavior of application entities when failures occur. The protocols used to achieve this behavior are described in Section 2.14.1, “Transaction Service Protocols,” on page 2-40.

From the viewpoint of each user object role, two types of failure are relevant: a failure affecting the object itself (local failure) and a failure external to the object (external failure), such as failure of another object or failure in the communication with that object.

2.13.12.1 Transaction Originator

Local Failure

A failure of a transaction originator prior to the originator issuing **commit** will cause the transaction to be rolled back. A failure of the originator after issuing **commit** and before the outcome is reported may result in either commitment or rollback of the transaction depending on timing; in this case completion of the transaction takes place without regard to the failure of the originator.

External Failure

Any external failure affecting the transaction prior to the originator issuing **commit** will cause the transaction to be rolled back; the standard exception TRANSACTION_ROLLEDBACK will be raised in the originator when it issues **commit**.

A failure after commit and before the outcome has been reported will mean that the client may not be informed of the transaction outcome, depending on the nature of the failure, and the use of the **report_heuristics** option of **commit**. For example, the transaction outcome will not be reported to the client if communication between the client and the coordinator fails.

A client may use **get_status** on the **Coordinator** to determine the transaction outcome. However, this is not reliable because the status **NoTransaction** is ambiguous: it could mean that the transaction committed and has been forgotten, or that the transaction rolled back and has been forgotten.

If an originator needs to know the transaction outcome, including in the case of external failures, then either the originator's implementation must include a **Resource** object so that it will participate in the two-phase commit procedure (and any recovery), or the originator and coordinator must be located in the same failure domain (for example, the same execution environment).

2.13.12.2 Transactional Server

Local Failure

If the Transactional Server fails, then optional checks by a Transaction Service implementation may cause the transaction to be rolled back. Without such checks, whether the transaction is rolled back depends on whether the commit decision has already been made. This would be the case where an unchecked client invokes **commit** before receiving all replies from servers.

External Failure

Any external failure affecting the transaction during the execution of a Transactional Server will cause the transaction to be rolled back. If this occurs while the transactional object's method is executing, the failure has no effect on the execution of this method. The method may terminate normally, returning the reply to its client. Eventually the TRANSACTION_ROLLEDBACK exception will be returned to a client issuing **commit**.

2.13.12.3 Recoverable Server

Behavior of a recoverable server when failures occur is determined by the two phase commit protocol between the coordinator and the recoverable server's **Resource** object(s). This protocol, including the local and external failure models and the required behavior of the Resource, is described in Section 2.14.1, "Transaction Service Protocols," on page 2-40.

2.14 The Implementers' View

This section contains three major categories of information.

1. Section 2.14.1, "Transaction Service Protocols," on page 2-40 defines in more detail the protocols of the Transaction Service for ensuring atomicity of transactions, even in the presence of failure.

   This section is *not* a formal part of the specification but is provided to assist in building valid implementations of the specification. These protocols affect implementations of Recoverable Servers and the Transaction Service.

2. Section 2.14.2, "ORB/TS Implementation Considerations," on page 2-51 provides additional information for implementers of ORBs and Transaction Services in those areas where cooperation between the two is necessary to realize the Transaction Service function.

   The following aspects of ORB and Transaction Service implementation are covered:
   - transaction propagation.
   - interoperation between different transaction service implementations.
   - ORB changes necessary to support portability of transaction service implementations.

3. Section 2.14.3, "Model Interoperability," on page 2-63 describes how an implementation achieves interoperation between the Transaction Service and procedural transaction managers.

2.14.1 Transaction Service Protocols

The Transaction Service requires that certain protocols be followed to implement the atomicity property. These protocols affect the implementation of recoverable servers, (recoverable objects that register for participation in the two-phase commit process) and the coordinators that are created by a transaction factory. These responsibilities ensure the execution of the two-phase commit protocol and include maintaining state information in stable storage, so that transactions can be completed in case of failures.

2.14.1.1 General Principles

The first coordinator created for a specific transaction is responsible for driving the two-phase commit protocol. In the literature, this is referred to as the *root Transaction Coordinator* or simply root coordinator. Any coordinator that is subsequently created for an existing transaction (for example, as the result of interposition) becomes a subordinate in the process. Such a coordinator is referred to as a *subordinate Transaction Coordinator* or simply subordinate coordinator and by registering a resource becomes a transaction participant. Recoverable servers are always transaction participants. The root coordinator initiates the two-phase commit protocol; participants respond to the operations that implement the protocol. The specification is based on the following rules for commitment and recovery:

1. The protocol defined by this specification is a two-phase commit with presumed rollback. This permits efficient implementations to be realized since the root coordinator does not need to log anything before the commit decision and the participants (i.e., **Resource** objects) do not need to log anything before they prepare.
2. **Resource** objects—including subordinate coordinators—do not start commitment by themselves, but wait for **prepare** to be invoked.
3. The **prepare** operation is issued at most once to each resource.
4. Participants must remember heuristic decisions until the coordinator or some management application instructs them to forget that decision.
5. A coordinator knows which **Resource** objects are registered in a transaction and so is aware of resources that have completed commitment. In general, the coordinator must remember this information if a transaction commits in order to ensure proper completion of the transaction. Resources can be forgotten early if they do not vote to commit the transaction.
6. A participant should be able to request the outcome of a transaction at any time, including after failures occurring subsequent to its **Resource** object being prepared.
7. Participants should be able to report the completion of the transaction (including any heuristic condition).

The recording of information relating to the transaction which is required for recovery is described as if it were a log file for clarity of description; an implementation may use any suitable persistent storage mechanism.

2.14.1.2 Normal Transaction Completion

Transaction completion can occur in two ways; as part of the normal execution of the **Current::commit** or **Terminator::commit** operations or independent of these operations if a failure should occur before normal execution can complete. This section describes the normal (no failure) case. Section 2.14.1.3, “Failures and Recovery,” on page 2-48 describes the failure cases.

Coordinator Role

The root coordinator implements the following protocol:

- When the client asks to **commit** the transaction, and no prior attempt to rollback the transaction has been made, the coordinator issues the **before_completion** request to all registered synchronizations.
- When all registered synchronizations have responded, the coordinator issues the **prepare** request to all registered resources.
- If all registered resources reply **VoteReadOnly**, then the root coordinator replies to the client that the transaction committed (assuming that the client can still be reached).

Before doing so, however, it first issues **after_completion** to any registered synchronizations and, after all responses are received, replies to the client. There is no requirement for the coordinator to log in this case.

- If any registered resource replies **VoteRollback** or cannot be reached, then the coordinator will decide to rollback and will so inform those registered resources that already replied **VoteCommit**.
- Once a **VoteRollback** reply is received, a coordinator need not send **prepare** to the remaining resources. **Rollback** will be subsequently sent to resources that replied **VoteCommit**.

  If the **report_heuristics** parameter was specified on **commit**, the client will be informed of the rollback outcome when any heuristic reports have been collected (and logged if required).
- Once at least one registered resource has replied **VoteCommit** and all others have replied **VoteCommit** or **VoteReadOnly**, a root coordinator may decide to commit the transaction.
- Before issuing **commit** operations on those registered resources that replied **VoteCommit**, the coordinator must ensure that the commit decision and the list of registered resources—those that replied **VoteCommit**—is stored in stable storage.
- If the coordinator receives **VoteCommit** or **VoteReadOnly** responses from each registered resource, it issues the **commit** request to each registered resource that responded **VoteCommit**.
- After having received all **commit** or **rollback** responses, if synchronizations exist, the root coordinator issues **after_completion** to each of them passing the transaction outcome as status before responding to the client.
- The root coordinator issues **forget** to a resource after it receives a heuristic exception.
- This responsibility is not affected by failure of the coordinator. When receiving commit replies containing heuristic information, a coordinator constructs a composite for the transaction.
- The root coordinator forgets the transaction after having logged its heuristic status if heuristics reporting was requested by the originator.
- The root coordinator can now trigger the sending of the reply to the **commit** operation if heuristic reporting is required. If no heuristic outcomes were recorded, the coordinator can be destroyed.

One Phase Commit

If a coordinator has only a single registered resource, it can perform the **commit_one_phase** operation on the resource instead of performing **prepare** and then **commit** or **rollback**. If a synchronization exists, **before_completion** is issued prior to **commit_one_phase** and **after_completion** is issued when the response to **commit_one_phase** has been received. If a failure occurs, the coordinator will not be informed of the transaction outcome.

Subtransactions

When completing a subtransaction, the subtransaction coordinator must notify any registered subtransaction aware resources of the subtransaction's commit or rollback status using the **commit_subtransaction** or **rollback_subtransaction** operations of the **SubtransactionAwareResource** interface.

A transaction service implementation determines how it chooses to respond when a resource responds to **commit_subtransaction** with a system exception. The service may choose to rollback the subtransaction or it may ignore the exceptional condition. The **SubtransactionAwareResource** operations are used to notify the resources of a subtransaction when the subtransaction commits in the case where the resource needs to keep track of the commit status of its ancestors. They are not used to direct the resources to commit or rollback any state. The operations of the **Resource** interface are used to commit or rollback subtransaction resources registered using the **register_resource** operation of the **Coordinator** interface.

When the subtransaction is committed and after all of the registered subtransaction aware resources have been notified of the commitment, the subtransaction registers any resources registered using **register_resource** with its parent **Coordinator** or it may register a subordinate coordinator to relay any future requests to the resources.

From the application programmer point of view, the same rules that apply to the completion of top-level transactions also apply to subtransactions. The **report_heuristics** parameter on **commit** is ignored since heuristics are not produced when subtransactions are committed.

Recoverable Server Role

A recoverable server includes at least one recoverable object and one **Resource** object. The recoverable object has state that demonstrates at least the atomicity property. The **Resource** object implements the two-phase commit protocol as a participant on behalf of the recoverable object. The responsibilities of each of these objects is described below.

Synchronization Registration

A recoverable server may need to register a **Synchronization** object to ensure that object state data, which is persistently managed by a resource is returned to the resource prior to starting the commitment protocol.

Top-Level Registration

A recoverable object registers a **Resource** object with the **Coordinator** so commitment of the transaction including any necessary recovery can be completed.

A recoverable object uses the **is_same_transaction** operation to determine whether it is already registered in this transaction. It can also use **hash_transaction** to reduce the number of comparisons. This relies on the definition of the **hash_transaction** operation to return the same value for all coordinators in the same transaction even if they are generated by multiple Transaction Service implementations.

Once registered, a recoverable server assumes the responsibilities of a transaction participant.

Subtransaction Registration

A Recoverable Server registers for subtransaction completion only if it needs to take specific actions at the time a subtransaction commits. An example would be to change ownership of locks acquired by this subtransaction to its parent.

A recoverable object uses the **is_same_transaction** operation to determine whether it is already registered in this subtransaction. It can also use **hash_transaction** to reduce the number of comparisons.

Top Level Synchronization

Synchronization objects ensure that persistent state data is returned to the recoverable object managed by a resource or to the underlying database manager. To do so they implement a protocol that moves the data prior to the prepare phase and does necessary processing after the outcome is complete.

Top-Level Completion

Resource objects implement a recoverable object's involvement in transaction completion. To do so, they must follow the two-phase commit protocol initiated by their coordinator and maintain certain elements of their state in stable storage. The responsibilities of a **Resource** object with regard to a particular transaction depend on how it will vote:

1. Returning **VoteCommit** to **prepare**

   Before a **Resource** object replies **VoteCommit** to a **prepare** operation, it must implement the following:

   - make persistent the recoverable state of its recoverable object.

     The method by which this is accomplished is implementation dependent. If a recoverable object has only transient state, it need not be made persistent.

   - ensure that its object reference is recorded in stable storage to allow it to participate in recovery in the event of failure.

     How object references are made persistent and then regenerated after a failure is outside the scope of this specification. The Persistent Object Service or some other mechanism may be used. How persistent **Resource** objects get restarted after a failure is also outside the scope of this specification.

   - record the **RecoveryCoordinator** object reference so that it can initiate recovery of the transaction later if necessary.
   - the **Resource** then waits for the coordinator to invoke **commit** or **rollback**.
   - A **Resource** with a heuristic outcome must not discard that information until it receives a **forget** from its coordinator or some administrative component.

2. Returning **VoteRollback** to **prepare**

A **Resource** that replies **VoteRollback** has no requirement to log. Once having replied, the **Resource** can return recoverable resources to their prior state and forget the transaction.

3. Returning **VoteReadOnly** to **prepare**

   A **Resource** that replies **VoteReadOnly** has no requirement to log. Once having replied, the **Resource** can release its resources and forget the transaction.

Subtransaction Completion

The role of the subtransaction aware resource at subtransaction completion are defined by the subtransaction aware resource itself. The coordinator only requires that it respond to **commit_subtransaction** or **rollback_subtransaction**.

All resources need to be notified when a transaction commits or is rolled back. But some resources need to know when subtransactions commit so that they can update local data structures and track the completion status of ancestors. The resource may have rules that are specific to ancestry and must perform some work as all or some ancestors complete. The nested semantics and effort required by the **Resource** object are defined by the object and not the Transaction Service.

Once the resource has been told to prepare, the resource's obligations are exactly the same as a top-level resource.

For example, in the Concurrency Control Service, a resource in a nested transaction might want to know when the subtransaction commits because another subtransaction may be waiting for a lock held by that subtransaction. Once that subtransaction commits, others may be granted the lock. There is no requirement to make lock ownership persistent until a **prepare** message is received.

For the Persistent Object Service, it is important to keep separate update information associated with a subtransaction. When that subtransaction commits, the Persistent Object Service may need to reorganize its information (such as undo information) in case the parent subtransaction chooses to rollback. Again, the Persistent Object Service resource need not make updates permanent until a **prepare** message is received. At that point, it has the same responsibilities as a top-level resource.

Subordinate Coordinator Role

An implementation of the Transaction Service may interpose subordinate coordinators to optimize the commit tree for completing the transaction. Such coordinators behave as transaction participants to their superiors and as coordinators to their resources or inferior coordinators.

Synchronization

A subordinate coordinator may register a **Synchronization** object with its superior coordinator if it needs to perform processing before its prepare phase begins.

Registration

A subordinate coordinator registers a **Resource** with its superior coordinator. Once registered, a subordinate coordinator assumes the responsibilities of a transaction participant and implements the behavior of a recoverable server.

Subtransaction Registration

If any of the resources registered with the subordinate coordinator support the **SubtransactionAwareResource** interface, the subordinate coordinator must register a subtransaction aware resource with its parent coordinator. If any of the resources registered with the subordinate using the **register_resource** operation, the subordinate must register a **Resource** with its superior. If both types of resources were registered with the subordinate, the subordinate only needs to register a subtransaction aware resource with its superior.

Top-level Completion

A subordinate coordinator implements the completion behavior of a recoverable server.

Subtransaction Completion

A subordinate coordinator implements the subtransaction completion behavior of a recoverable server.

Subordinate Coordinator

A subordinate coordinator does not make the commit decision but simply relays the decision of its superior (which may also be a subordinate coordinator) to resources registered with it. A subordinate coordinator acts as a recoverable server as described previously, in terms of saving its state in stable storage. A subordinate coordinator (or indeed any resource) may log the commit decision once it is known (as an optimization) but this is not essential.

- A subordinate coordinator issues the **before_completion** operation to any synchronizations when it receives **prepare** from its superior.
- When all responses to **before_completion** have been received, a subordinate coordinator issues the **prepare** operation to its registered resources.
- If all registered resources reply **VoteReadOnly**, then the subordinate coordinator will decide to reply **VoteReadOnly**.

  Before doing so, however, it first issues **after_completion** to any registered synchronizations and, after all responses are received, replies **VoteReadOnly** to its superior. There is no requirement for the subordinate coordinator to log in this case; the subordinate coordinator takes no further part in the transaction and can be destroyed.
- If any registered resource replies **VoteRollback** or cannot be reached, then the subordinate coordinator will decide to rollback and will so inform those registered resources that already replied **VoteCommit**.

Once a **VoteRollback** reply is received, the subordinate coordinator need not send **prepare** to the remaining resources. The subordinate coordinator issues **after_completion** to any synchronizations and, after all responses have been received, replies **VoteRollback** to its superior.

- Once at least one registered resource has replied **VoteCommit** and all others have replied **VoteCommit** or **VoteReadOnly**, a subordinate coordinator may decide to reply **VoteCommit**.

  The subordinate coordinator must record the prepared state, the reference of its superior **RecoveryCoordinator** and its list of resources that responded **VoteCommit** in stable storage before responding to **prepare**.

- A subordinate coordinator issues the **commit** operation to its registered resources, which replied **VoteCommit** when it receives a **commit** request from its superior.
- If any resource reports a heuristic outcome, the subordinate coordinator must report a heuristic outcome to its superior.

  Before doing so, however, it first issues **after_completion** to any registered synchronizations and, after all responses are received, reports the heuristic outcome to its superior. The specific outcome reported depends on the other heuristic outcomes received. The subordinate coordinator must record the heuristic outcome in stable storage.

- After having received all **commit** replies, a subordinate coordinator logs its heuristic status (if any).
- The subordinate coordinator then replies to the **commit** from its superior coordinator.

  Before doing so, it issues **after_completion** to any registered synchronizations and, after all responses have been received, it then replies to its superior. If no heuristic report was sent the **Coordinator** is destroyed.

- A subordinate coordinator performs the **rollback** operation on its registered resources when it receives a **rollback** request from its superior.

  If any resource reports a heuristic outcome, the subordinate coordinator records the appropriate heuristic outcome in stable storage and will report this outcome to its superior. Before doing so, however, it issues **after_completion** to any registered synchronizations and, after receiving all the responses, reports the heuristic outcome to its superior.

- The subordinate coordinator then replies to the **rollback** from its superior coordinator.

  Before doing so, it issues **after_completion** to any registered synchronizations and, after all responses have been received, it then replies to its superior. If no heuristic report was sent the **Coordinator** is destroyed.

- If a subordinate coordinator receives a **commit_one_phase** request, and it has a single registered resource, it can simply perform the **commit_one_phase** request on its resource. Before doing so, if a synchronization exists, it issues **before_completion** to the synchronization, then, after receiving the **commit_one_phase** response, issues **after_completion** to the synchronization.

  If it has multiple registered resources, it behaves like a superior coordinator, issuing **before_completion** to any synchronizations and, after receiving the responses, issuing **prepare** to each resource to determine the outcome, then issuing **commit** or **rollback** requests, followed by **after_completion** requests if synchronizations exist.
- A subordinate coordinator performs the **forget** operation on those registered resources that reported a heuristic outcome when it receives a **forget** request from its superior.

Subtransactions

A subordinate coordinator for a subtransaction relays **commit_subtransaction** and **rollback_subtransaction** requests to any subtransaction aware resources registered with it. In addition, it performs the same roles as a top-level subordinate coordinator when the top-level transaction commits. It must relay **prepare** and **commit** requests to each of the resources that registered with it using the **register_resource** operation.

2.14.1.3 *Failures and Recovery*

The previous descriptions dealt with the protocols associated with the Transaction Service when a transaction completes without failure. To ensure atomicity and durability in the presence of failure, the transaction service defines additional protocols to ensure that transactions, once begun, always complete.

Failure Processing

The unit of failure is termed the failure domain. It may consist of the coordinator and some local resources registered with it, or the coordinator and the resources may each be in its own failure domain.

Local Failure

Any failure in the transaction during the execution of a coordinator prior to the commit decision being made will cause the transaction to be rolled back.

A coordinator is restarted only if it has logged the commit decision.

- If the coordinator only contains heuristic information, nothing is done.
- If the transaction is marked rollback only, a coordinator can send **rollback** to its resources and inferior coordinators.
- If the transaction outcome is commit, the coordinator sends **commit** to prepared registered resources and the regular commitment procedure is started.
- If any registered resources exist but cannot be reached, then the coordinator must try again later.

If registered resources no longer exist, then this means that they completed commitment before the coordinator failed and have no heuristic information.

- If a subordinate coordinator is prepared, then it must contact its superior coordinator to determine the transaction outcome.
- If the superior coordinator exists but cannot be reached, then the subordinate must retry recovery later.
- If the superior coordinator no longer exists, then the outcome of the transaction can be presumed to be rollback.

The subordinate will inform its registered resources.

External Failure

Any failure in the transaction during the execution of a coordinator prior to the commit decision being made will cause the transaction to be rolled back.

2.14.1.4 Transaction Completion after Failure

In general, the approach is to continue the completion protocols at the point where the failure occurred. That means that the coordinator will usually have the responsibility for sending the commit decision to its registered resources. Certain failure conditions will require that the resource initiate the recovery procedure—recall that the resource might also be a subordinate coordinator. These are described in more detail below.

Resources

A resource represents some collection of recoverable data associated with a transaction. It supports the **Resource** interface described in Section 2.8, “Resource Interface,” on page 2-14. When recovering from failure after its changes have been prepared, a resource uses the **replay_completion** operation on the **RecoveryCoordinator** to determine the outcome of the transaction and continue completion.

Heuristic Reporting

If the coordinator does not complete the two-phase commit in a timely manner, a subordinate (i.e., a resource or a subordinate coordinator) in the transaction may elect to commit or rollback the resources registered with it in a prepared transaction (take a ***heuristic decision***). When the coordinator eventually sends the outcome, the outcome may differ from that heuristic decision. The result is referred to as **HeuristicMixed** or **HeuristicHazard**. The result is reported by the root coordinator to the client only when the **report_heuristics** option on **commit** is selected. In these circumstances, the participant (subordinate) and the coordinator must obey a set of rules that define what they report.

Coordinator Role

A root coordinator that fails prior to logging the commit decision can unilaterally rollback the transaction. If its resources have also rolled back because they were not prepared, the transaction is returned to its prior state of consistency. If any resources are prepared, they are required to initiate the recovery process defined below.

- A root coordinator that has a committed outcome will continue the completion protocol by sending **commit**.
- A root coordinator that has a rolled back outcome will continue the completion protocol by sending **rollback**.

Synchronizations

Synchronization objects are not persistent so they are not restarted after failure and, as a result, their operations are not invoked during failure processing.

Subtransactions

Subtransactions are not durable, so there is no completion after failure. However, once the top-level coordinator issues **prepare**, a subtransaction subordinate coordinator has the same responsibilities as a top-level subordinate coordinator.

Recoverable Server role

The Transaction Service imposes certain requirements on the recoverable objects participating in a transaction. These requirements include an obligation to retain certain information at certain times in stable storage (storage not likely to be damaged as the result of failure). When a recoverable object restarts after a failure, it participates in a recovery protocol based on the contents (or lack of contents) of its stable storage.

Once having replied **VoteCommit**, the resource remains responsible for discovering the outcome of the transaction (i.e., whether to commit or rollback). If the resource subsequently makes a heuristic decision, this does not change its responsibilities to discover the outcome.

If No Heuristic Decision is Made

A resource that is prepared is responsible for initiating recovery. It does so by issuing **replay_completion** to the **RecoveryCoordinator**. The reply tells the resource the outcome of the transaction. The coordinator can continue the completion protocol allowing the resource to either commit or rollback. The resource can resend **replay_completion** if the completion protocol is not continued.

- If the resource having replied **VoteCommit** initiates recovery and receives **StExcep::OBJECT_NOT_EXIST**, it will know that the **Coordinator** no longer exists and therefore the outcome was to rollback (presumed rollback).
- If the resource having replied **VoteCommit** initiates recovery and receives **StExcep::COMM_FAILURE**, it will know only that the **Coordinator** may or may not exist. In this case, the resource retains responsibility for initiating recovery again at a later time.

When a Heuristic Decision is Made

Before acting on a heuristic decision, it must record the decision in stable storage.

- If the heuristic decision turns out to be consistent with the outcome, then all is well and the transaction can be completed and the heuristic decision can be forgotten.
- If the heuristic decision turns out to be wrong, the heuristic damage is recorded in stable storage and one of the heuristic outcome exceptions (**HeuristicCommit**, **HeuristicRollback**, **HeuristicMixed**, or **HeuristicHazard**) is returned when completion continues.

The heuristic outcome details must be retained persistently until the resource is instructed to forget. In this case, the resource remains persistent until the **forget** is received.

Subordinate Coordinator Role

The behavior of a subordinate coordinator after a failure of its superior coordinator is implementation-dependent; however, it does follow the following protocols:

- Since it appears as a resource to its superior coordinator, the protocol defined for recoverable servers applies to subordinate coordinators.
- Since it is also a subordinate coordinator for its own registered resources, it is permitted to send duplicate **commit**, **rollback**, and **forget** requests to its registered resources.
- It is required to (eventually) perform either **commit** or **rollback** on any resource to which it has received a **VoteCommit** response to **prepare**.
- It[1] is required to (eventually) perform the **forget** operation on any resource that reported a heuristic outcome.

Since subtransactions are not durable, it has no responsibility in this area for failure recovery.

2.14.2 ORB/TS Implementation Considerations

The Transaction Service and the ORB must cooperate to realize certain Transaction Service function. This cooperation is realized on the **client invocation path** and through the **transaction interceptor**. The client invocation path is present even in an OTS-unaware ORB and is required to make certain checks to ensure successful interoperability. The transaction interceptor is a request-level interceptor that is bound into the invocation path. This cooperation is discussed in greater detail in the following sections.

1. or some “agent” acting on its behalf: for example a system management application.

2.14.2.1 Policy Checking Requirements

This section describes the policy checks that are required on the client side before a request is sent to a target object and the server side when a request is received. The client invocation path is used to describe components of the client-side ORB which may include the ORB itself, the generated client stub, CORBA messaging, and the OTS interceptor. This function will be more rigorously assigned to each of these components in a future revision of the OTS specification. The server side includes the server-side ORB, the POA, and the OTS interceptor.

Client behavior when making transactional invocations

When a client makes a request on a target object, the behavior is influenced by the type of invocation, the existence of an active client transaction, and the **InvocationPolicy** and **OTSPolicy** associated with the target object. The client invocation path must verify that the client invocation mode matches the requirements of the target object. This requires checking the **InvocationPolicy** encoded in the IOR and, in some cases, the **OTSPolicy**. The required behavior is completely described by the following tables.

Table 2-8 InvocationPolicy checks required on the client invocation path

Invocation Mode	InvocationPolicy	Required Action
Synchronous	EITHER	ok; check **OTSPolicy**
	SHARED	ok; check **OTSPolicy**
	UNSHARED	raise **TRANSACTION_MODE**
Asynchronous	EITHER	ok; check **OTSPolicy**
	SHARED	raise **TRANSACTION_MODE**
	UNSHARED	ok; check **OTSPolicy**

An invocation is considered synchronous if it uses a standard client stub, the DII, or AMI with an effective routing policy of **ROUTE_NONE**. An invocation is considered asynchronous if it uses the features of CORBA messaging to invoke on a router rather than the target object.

Table 2-9 OTSPolicy checks required on the Client Invocation Path

OTSPolicy	OTS-unaware ORB	OTS-aware ORB
REQUIRES	raise TRANSACTION_UNAVAILABLE	call OTS interceptor
FORBIDS	process invocation	call OTS interceptor
ADAPTS	process invocation	call OTS interceptor

In the case of routed invocations, the client invocation path must substitute an appropriate router IOR before the **OTSPolicy** checks are executed. This ensures that the **OTSPolicy** checks are done against the correct IOR.

The client OTS interceptor is required to make the following policy checks before processing the transaction context. Transaction context processing is described in Section 2.14.2.5, “Behavior of the Callback Interfaces,” on page 2-61.”

Table 2-10 OTSPolicy checking required by client OTS interceptor

OTSPolicy	Current Transaction	No Current Transaction
REQUIRES	process transaction	raise TRANSACTION_REQUIRED
FORBIDS [1]	PREVENT - raise INVALID_TRANSACTION PERMIT - process transaction	process invocation
ADAPTS	process transaction	process invocation

[1] FORBIDS processing depends on the setting of the **NonTxTargetPolicy** policy.

Server-side behavior when receiving transactional invocations

Since the active transaction state as seen by the server-side can be different than the state observed by the client ORB, the server-side is also required to make the **OTSPolicy** checks. These checks will be made prior to the service context propagation checks defined in Section 2.14.2.5, “Behavior of the Callback Interfaces,” on page 2-61.

Table 2-11 OTSPolicy checks required on the Server-side

OTSPolicy	OTS-unaware ORB	OTS-aware ORB
REQUIRES	raise TRANSACTION_UNAVAILABLE	call OTS interceptor
FORBIDS	process invocation	call OTS interceptor
ADAPTS	process invocation	call OTS interceptor

The server OTS interceptor is required to make the following policy checks before processing the transaction context. Transaction context processing is described in Section 2.14.2.5, “Behavior of the Callback Interfaces,” on page 2-61.”

Table 2-12 OTSPolicy checking required by server OTS interceptor

OTSPolicy	Current Transaction	No Current Transaction
REQUIRES	process transaction	raise TRANSACTION_REQUIRED
FORBIDS	raise INVALID_TRANSACTION	process invocation
ADAPTS	process transaction	process invocation

Alternate Client processing for FORBIDS OTSPolicy component

When the **NonTxTargetPolicy** policy is set to PERMIT, the processing of the FORBIDS value (whether it is explicitly encoded as a **TAG_OTS_POLICY** component or determined by the absence of inheritance from **TransactionalObject**) does not raise the INVALID_TRANSACTION exception. Instead it is altered as described below.

Since an OTS must be present for a client to have a current transaction at the time an invocation is made, the client OTS interceptors must also be present within the client environment. This permits an alternative behavior to be implemented on the client-side which maintains compatibility with prior versions of OTS and simplifies client programming when making invocations on non-transactional objects. This alternative behavior is summarized below:

- When the target object supports the FORBIDS policy, the alternative behavior is implemented if the **NonTxTargetPolicy** policy is set to PERMIT.
- The client-side request interceptor must ensure that the current transaction is inactive before the transaction propagation checks are executed (Section 2.14.2.5, “Behavior of the Callback Interfaces,” on page 2-61).
- The current transaction must be made active after the request has successfully executed.

The current transaction can be made inactive by performing the equivalent of a **suspend** operation on the current transaction prior to implementing the transaction propagation rules and made active again by performing the equivalent of a **resume** operation when the response is returned to restore the client’s current transaction. An implementation that produces equivalent results but does not use the **suspend** and **resume** operation defined by this specification is conformant.

This preserves the client programming model of earlier OTS levels while still guaranteeing that transactions will not be exported to environments that do not understand transactional semantics.

Interoperation with OTS 1.1 servers and clients

When OTS 1.2 clients are interoperating with OTS 1.1 servers (i.e., the IOR does not contain **TAG_OTS_POLICY** component) the client invocation path must determine if the target object inherits from **TransactionalObject**. If it does, it processes the request as if the **OTSPolicy** value was ADAPTS. If it does not, it processes the request as if the **OTSPolicy** value was FORBIDS and uses the **NonTxTargetPolicy** policy to determine the correct behavior.

OTS 1.1 clients may not interoperate with OTS 1.2 servers unless they unconditionally propagate the transaction context. The OTS 1.2 server determines the proper **OTSPolicy** from the **TAG_OTS_POLICY** component in the IOR.

An OTS 1.2 object that also inherits from the deprecated **TransactionalObject** (for backward compatibility) must create POAs with a **OTSPolicy** value of REQUIRES or ADAPTS - any other policy value is illegal and is an implementation error.

2.14.2.2 Transaction Propagation

The transaction is represented to the application by the **Control** object. Within the Transaction Service, an implicit context is maintained for all threads associated with a transaction. Although there is some common information, the implicit context is not the same as the **Control** object defined in this specification and is distinct from the ORB Context defined by CORBA. It is the implicit context that must be transferred between execution environments to support transaction propagation.

The objects using a particular Transaction Service implementation in a system form a Transaction Service domain. Within the domain, the structure and meaning of the implicit context information can be private to the implementation. When leaving the domain, this information must be translated to a common form if it is to be understood by the target Transaction Service domain, even across a single ORB. When the implicit context is transferred, it is represented as a **PropagationContext**.

No OMG IDL declaration is required to cause propagation of the implicit context with a request. The minimum amount of information that could serve as an implicit context is the object reference of the **Coordinator**. However, an identifier (e.g., an X/Open XID) is also required to allow efficient (local) execution of the **is_same_transaction** and **hash_transaction** operations when interposition is done. Implementations may choose to also include the **Terminator** object reference if they support the ability for ending the transaction in other execution environments than the originator's. Transferring the implicit context requires interaction between the Transaction Service and the ORB to add or extract the implicit context from ORB messages. This interaction is also used to implement the checking functions described in Section 2.13.4, "X/Open Checked Transactions," on page 2-29.

When the **Control** object is passed as an operation argument (explicit propagation), no special transfer mechanism is required.

Interposition

When a transaction is propagated, the implicit context is exported and can be used by the importing Transaction Service implementation to create a new **Control** object, which refers to a new (local) **Coordinator**. This technique, *interposition*, allows a surrogate to handle the functions of a coordinator in the importing domain. These coordinators act as subordinate coordinators. When interposition is performed, a single transaction is represented by multiple **Coordinator** objects.

Interposition allows cooperating Transaction Services to share the responsibility for completing a transaction and can be used to minimize the number of network messages sent during the completion process. Interposition is required for a Transaction Service implementation to implement the **is_same_transaction** and **hash_transaction** operations as local method invocations, thus improving overall systems performance.

An interposed coordinator registers as a participant in the transaction with the **Coordinator** identified in the **PropagationContext** of the received request. The relationships between coordinators in the transaction form a tree. The root coordinator is responsible for completing the transaction.

Many implementations of the Transaction Service will want to perform interposition and thus create **Control** objects and subsequently **Coordinator** objects for each execution environment participating in the transaction. To create a new (local) **Control**, an importing Transaction Service uses the information in the propagation context to **recreate** a **Control** object using a **TransactionFactory**. Interposition must be complete before the **get_control** operation can complete in the target object. An object adapter is one possible place to implement interposition.

Subordinate Coordinator Synchronization

A subordinate coordinator may register with its superior coordinator to ensure that any local state data maintained by the subordinate coordinator is returned to the underlying resource prior to the subordinate coordinator's associated **Resource** seeing **prepare**.

Subordinate Coordinator Registration

A subordinate coordinator must register with its superior coordinator to orchestrate transaction completion for its local resources. The **register_resource** operation of the **Coordinator** can be used to perform this function. The subordinate coordinator can either support the **Resource** interface itself or provide another **Resource** object that will support transaction completion. Some implementations of the Transaction Service may wish to perform this function as a by-product of invoking the first operation on an object in a new domain as is done with the X/Open model. This requires that the information necessary to perform registration be added to the reply message of that first operation.

2.14.2.3 Transaction Service Interoperation

The Transaction Service can be implemented by multiple components at different locations. The different components can be based on the same or different implementations of the Transaction Service. As stated in Section 1.2.5, "Principles of Function, Design, and Performance," on page 1-8, it is a requirement that multiple Transaction Services interoperate across the same ORB and different ORBs.

Transaction Service interoperation is specified by defining the data structures exported between different implementations of the Transaction Service. When the implicit context is propagated with a request, the destination uses it to locate the superior coordinator. That coordinator may be implemented by a foreign Transaction Service. By registering a resource with that coordinator, the destination arranges to receive two-phase commit requests from the (possibly foreign) Transaction Service.

The Transaction Service permits many configurations; no particular configuration is mandated. Typically, each program will be directly associated with a single Transaction Service. However, when requests are transmitted between programs in different Transaction Service domains, both Transaction Services must understand the shared data structures to interoperate.

An interface between the ORB and the Transaction Service is defined that arranges for the implicit context to be carried on messages that represent method invocations made within the scope of a transaction.

Structure of the Propagation Context

The **PropagationContext** structure is defined in Section 1.3.5, “Structures,” on page 1-16. It is passed between Transaction Service domains as an **IOP::ServiceContext** in both GIOP requests and replies. Implementations may use the vendor specific portion for additional functions (for example, to register an interposed coordinator with its superior).

otid_t

The **otid_t** structure is a more efficient OMG IDL version of the X/Open defined transaction identifier (XID). The **otid_t** can be transformed to an X/Open XID and vice versa.

TransIdentity

A structure that defines information for a single transaction. It consists of a **coord**, an optional **term**, and an **otid**.

coord

The **Coordinator** for this transaction in the exporting Transaction Service domain.

term

The **Terminator** for this transaction in the exporting Transaction Service domain. Transaction Services that do not allow termination by other than the originator will set this field to a null reference (**OBJECT_NIL**).

otid

An identifier specific to the current transaction or subtransaction. This value is intended to support efficient (local) execution of the **is_same_transaction** and **hash_transaction** operations when the importing Transaction Service does interposition.

timeout

The timeout value associated with the hierarchy’s top-level transaction in the relevant **set_timeout** operation (or the default timeout). This timeout is the time remaining, i.e. the timeout when the transaction was begun (or the timeout received through a transactional request, or through a **Current::resume** operation) less the elapsed time, in seconds.

<TransIdentity> parents

A sequence of **TransIdentity** structures representing the parent(s) of the current transaction. The ordering of the sequence starts at the parent of the current transaction and includes all ancestors up to the top-level transaction. An implementation that does not support nested transactions would send an empty sequence. This allows a non-

nested transaction implementation to know when a nested transaction is being imported. It also supports efficient (local) execution of the **Coordinator** operations which test parentage when the importing Transaction Service does interposition.

implementation_specific_data

This information is exported from an implementation and is required to be passed back with the rest of the context if the transaction is re-imported into that implementation. The intent is to permit additional information to be sent that might optimize the commit process (e.g., the entire transaction tree rather than just the immediate ancestors). In the case of OTS interoperation across any vendor boundaries, the importing implementation must not require that any specific information is passed as part of the **implementation_specific_data**. It must only pass back the provided information to the exporting implementation.

Appearance of the Propagation Context in Messages

The appearance of the **PropagationContext** in messages is defined by the CORBA interoperability specification (see the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*). The Transaction Service passes the **PropagationContext** to the ORB via the **TSPortability** interface defined in “The Transaction Service Callbacks” on page 2-60.

- When exporting a transaction, the ORB sets the **PropagationContext** into the **ServiceContext::context_data** field and marshals the **PropagationContext** as defined by the GIOP message format and marshalling rules.
- When importing a transaction, the ORB demarshalls the **ServiceContext::context_data** according to the GIOP formatting rules and extracts the **PropagationContext** to be presented to the Transaction Service.

For more information, see the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*.

2.14.2.4 Transaction Service Portability

This section describes the way in which the ORB and the Transaction Service cooperate to enable the **PropagationContext** to be passed and any X/Open-style checking to be performed on transactional requests.

Because it is recognized that other object services and future extensions to the CORBA specification may require similar mechanisms, this component is specified separately from the main body of the Transaction Service to allow it to be revised or replaced by a mechanism common to several services independently of any future Transaction Service revisions.

To enable a single Transaction Service to work with multiple ORBs, it is necessary to define a specific interface between the ORB and the Transaction Service, which conforming ORB implementations will provide, and demanding Transaction Service implementations can rely on. The remainder of this section describes these interfaces. There are two elements of the required interfaces:

1. An additional ORB interface that allows the Transaction Service to identify itself to the ORB when present in order to be involved in the transmission of transactional requests.
2. A collection of Transaction Service operations (the Transaction Service callbacks) that the ORB invokes when a transactional request is sent and received.

These interfaces are defined as pseudo-IDL to allow them to be implemented as procedure calls.

Identification of the Transaction Service to the ORB

Prior to the first transactional request, the Transaction Service will identify itself to the ORB within its domain to establish the transaction callbacks to be used for transactional requests and replies.

The Transaction Service identifies itself to the ORB using the following interface.

```
interface TSIdentification { // PIDL
    exception NotAvailable {};
    exception AlreadyIdentified {};

    void identify_sender(in CosTSPortability::Sender sender)
        raises (NotAvailable, AlreadyIdentified);
    void identify_receiver(in CosTSPortability::Receiver receiver)
        raises (NotAvailable, AlreadyIdentified);
};
```

The callback routines identified in this operation are always in the same addressing domain as the ORB. On most machine architectures, there are a unique set of callbacks per address space. Since invocation is via a procedure call, independent failures cannot occur.

NotAvailable

The NotAvailable exception is raised if the ORB implementation does not support the **CosTSPortability** module.

AlreadyIdentified

The AlreadyIdentified exception is raised if the **identify_sender** or **identify_receiver** operation had previously identified callbacks to the ORB for this addressing domain.

identify_sender

The **identify_sender** operation provides the interface that defines the callbacks to be invoked by the ORB when a transactional request is sent and its reply received.

identify_receiver

The **identify_receiver** operation provides the interface that defines the callbacks to be invoked by the ORB when a transactional request is received and its reply sent.

The Transaction Service must identify itself to the ORB at least once per Transaction Service domain. Sending and receiving transactional requests are separately identified. If the callback interfaces are different for different processes within a Transaction Service domain, they are identified to the ORB on a per process basis. Only one Transaction Service implementation per addressing domain can identify itself to the ORB.

A Transaction Service implementation that only sends transactional request can identify only the sender callbacks. A Transaction Service that only receives transactional requests can identify only the receiver callbacks.

The Transaction Service Callbacks

The **CosTSPortability** module defines two interfaces. Both interfaces are defined as PIDL. The **Sender** interface defines a pair of operations, which are called by the ORB sending the request before it is sent and after its reply is received. The **Receiver** interface defines a pair of operations that are called by the ORB receiving the request when the request is received and before its reply is sent. Both interfaces use the **PropagationContext** structure defined in Section 1.3.5, "Structures," on page 1-16.

```
module CosTSPortability { // PIDL
    typedef long ReqId;

    interface Sender {
        void sending_request(in ReqId id,
            out CosTransactions::PropagationContext ctx);
        void received_reply(in ReqId id,
            in CosTransactions::PropagationContext ctx,
            in CORBA::Environment env);
    };

    interface Receiver {
        void received_request(in ReqId id,
            in CosTransactions::PropagationContext ctx);
        void sending_reply(in ReqId id,
            out CosTransactions::PropagationContext ctx);
    };
};
```

ReqId

The **ReqId** is a unique identifier generated by the ORB, which lasts for the duration of the processing of the request and its associated reply to allow the Transaction Service to correlate callback requests and replies.

Sender::sending_request

A request is about to be sent. The Transaction Service returns a **PropagationContext** to be delivered to the Transaction Service at the server managing the target object. A null **PropagationContext** is returned if invoked outside the scope of a transaction.

Sender::received_reply

A reply has been received. The **PropagationContext** from the server is passed to the Transaction Service along with the returned environment. The Transaction Service examines the Environment to determine whether the request was successfully performed. A request completes unsuccessfully if it raises a system exception. Requests that raise a user exception or no exception at all are deemed to have completed successfully. Requests that are deemed unsuccessful cause the transaction associated with the request to be marked rollback only. This ensures that a subsequent call to commit will raise the TRANSACTION_ROLLBACKED system exception.

Receiver::received_request

A request has been received. The **PropagationContext** defines the transaction making the request.

Receiver::sending_reply

A reply is about to be sent. A checking transaction service determines whether there are outstanding deferred requests or subtransactions and raises a system exception using the normal mechanisms. The exception data from the callback operation needs to be re-raised by the calling ORB.

2.14.2.5 Behavior of the Callback Interfaces

The following describes the behavior of the ORB and Transaction Service in managing the callback interfaces. The behavior is based on a combination of an active connection between the transaction service and the ORB and the presence or absence of a transaction service context in the GIOP message. The new behavior is summarized below:

Client sending a Request

When the client ORB sends a request, there are three possible transaction service states in the client:

- **OTS_NOT_CONNECTED** - The transaction service has not connected to the client ORB. In this state, the client ORB does not invoke the **Sending_Request** operation and no transaction service context is inserted in the GIOP request message.
- **OTS_NO_CURRENT_TRANSACTION** - The transaction service has connected to the client ORB, but there is no **Current** transaction associated with the client's request. In this state, the client ORB invokes the **Sending_Request** operation and the transaction service returns a null **PropagationContext**. The client ORB does **not** place a transaction service context in the GIOP request message.

- **OTS_CURRENT_TRANSACTION** - The transaction service is connected to the client ORB and there is a **Current** transaction associated with the client's request. In this state, the client ORB invokes the **Sending_Request** operation and receives a **PropagationContext** from the transaction service. The **PropagationContext** is inserted into the transaction service context of the GIOP request message.

The client ORB cannot distinguish between states 2 and 3 and knows both as OTS (a transaction service is connected to the ORB). This difference is known by the transaction service, which implements the difference in behavior.

Server Receiving a Request

The server ORB receiving a request has two transaction service states:

- OTS_NOT_CONNECTED - as defined for the client, and
- OTS - a transaction service is connected to the server ORB.

Additionally the server ORB has two states defined by the presence or absence of a transaction service context in the GIOP request message. The server ORB behavior is captured below:

- If no transaction service context is present in the GIOP request message, the server ORB does **not** call the **Receiving_Request** operation and sets NO_REPLY to TRUE. This will be tested when the reply is ready to be sent.
- If a transaction service context is present in the GIOP request message and the transaction service state is OTS_NOT_CONNECTED, the server ORB raises the TRANSACTION_UNAVAILABLE exception back to the client and does not deliver the method request.
- If a transaction service context is present and the transaction service state is OTS, the server ORB invokes **Receiving_Request** passing the transaction service context to the server ORB's transaction service as a **PropagationContext**.

Server sending a Reply

The server ORB sending a reply is driven by the NO_REPLY state set by receiving this request and the transaction service state. Its behavior is as follows:

- If NO_REPLY is TRUE for this reply (there can be multiple outstanding with deferred synchronous), then the server ORB does **not** call **Sending_Reply** and does **not** insert a service context in the GIOP reply message.
- If NO_REPLY is FALSE and the transaction service state is OTS_NOT_CONNECTED, the server ORB raises the TRANSACTION_ROLLEDBACK exception back to the client. The client is then required to either initiate **Rollback** or mark the transaction **rollback_only**. This can only happen if the transaction service abnormally terminates between the time the request is received and the reply is ready to be sent.
- If NO_REPLY is FALSE and the transaction service state is OTS, invoke **Sending_Reply** and insert the returned **PropagationContext** in the transaction service context of the GIOP reply message.

Client Receiving a Reply

A client ORB receiving a reply is driven by the presence or absence of a transaction service context in the GIOP reply message and the two transaction service states (OTS and OTS_NOT_CONNECTED). The behavior is outlined below:

- If a transaction service context is not present in the GIOP reply message, the client ORB does **not** call **Receiving_Reply**.
- If a transaction service context is present in the GIOP reply message and the transaction service state is OTS_NOT_CONNECTED, the client ORB raises the TRANSACTION_ROLLEDBACK exception back to the client. Like it's analog in the server, this can only happen if the client transaction service abnormally terminates between the time the request is sent and the reply is received. Since the client's transaction service is no longer active, subsequent operations on any of the OTS interfaces will fail (OBJECT_NOT_EXIST) and the in-flight transaction will rollback when the transaction service is subsequently restarted.
- If a transaction service context is present in the GIOP reply message and the transaction service state is OTS, the client ORB invokes **Receiving_Reply** passing the transaction service context as a **PropagationContext**.

2.14.3 Model Interoperability

The indirect context management programming model of the Transaction Service is designed to be compatible with the X/Open DTP standard, and implementable by existing Transaction Managers. In X/Open DTP, a current transaction is associated with a ***thread of control***. Some X/Open Transaction Managers support a single thread of control in a ***process***, others allow multiple threads of control per process.

Model interoperability is possible because the Transaction Service design is compatible with the X/Open DTP model of a Transaction Manager. X/Open associates an implicit current transaction with each thread of control.

This means that a single transaction management service can provide the interfaces defined for the Transaction Service and also provide the TX and XA interfaces of X/Open DTP. This is illustrated in Figure 2-4.

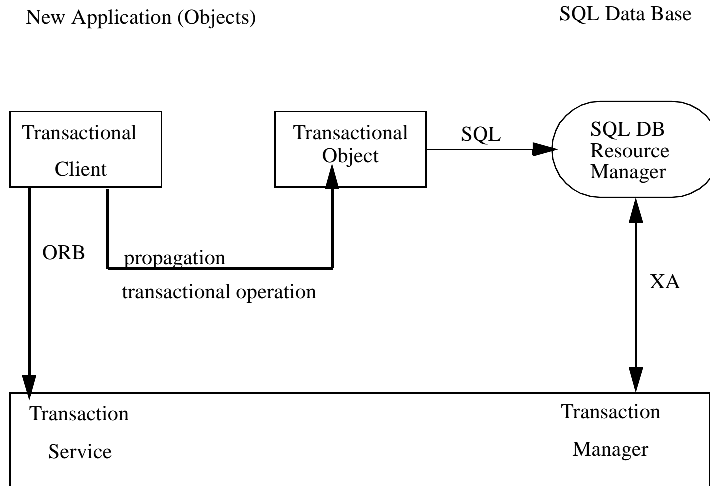

Figure 2-4 Model Interoperability Example

The transactional object making the SQL call, and the SQL Resource manager, are both executing on the same thread of control. The transaction manager is able to recognize the relationship between the transaction context of the object, and the transaction associated with the SQL DB.

The **Current** and **Coordinator** interfaces of the Transaction Service implement two-phase commit for the objects in the transaction. The Resource Manager will participate in the two-phase commitment process via the X/Open XA interface.

Complete OMG IDL A

Note – All text in black is from the Transaction Service, v1.2 (formal/01-11-03). Text in blue is from the Components specification.

A.1 The CosTransactions Module

```
#include <orb.idl>
module CosTransactions {
// DATATYPES
enum Status {
    StatusActive,
    StatusMarkedRollback,
    StatusPrepared,
    StatusCommitted,
    StatusRolledBack,
    StatusUnknown,
    StatusNoTransaction,
    StatusPreparing,
    StatusCommitting,
    StatusRollingBack
};

enum Vote {
    VoteCommit,
    VoteRollback,
    VoteReadOnly
};

typedef unsigned short TransactionPolicyValue;
// TransactionPolicyValue definitions are deprecated and replaced //
// with new InvocationPolicy and OTSPolicy definitions. They are //
```

```
// retained for backward compatibility. //
const TransactionPolicyValue    Allows_shared = 0;
const TransactionPolicyValue    Allows_none = 1;
const TransactionPolicyValue    Requires_shared = 2;
const TransactionPolicyValue    Allows_unshared = 3;
const TransactionPolicyValue    Allows_either = 4;
const TransactionPolicyValue    Requires_unshared = 5;
const TransactionPolicyValue    Requires_either = 6;

// Forward references for interfaces defined later in module
local interface Current;
interface TransactionFactory;
interface Control;
interface Terminator;
interface Coordinator;
interface RecoveryCoordinator;
interface Resource;
interface Synchronization;
interface SubtransactionAwareResource;

// TransactionalObject has been deprecated.
interface TransactionalObject;

// Structure definitions
struct otid_t {
      long formatID; /*format identifier. 0 is OSI TP */
      long bqual_length;
      sequence <octet> tid;
};

struct TransIdentity {
      Coordinator coord;
      Terminator term;
      otid_t otid;
};
struct PropagationContext {
      unsigned long timeout;
      TransIdentity current;
      sequence <TransIdentity> parents;
      any implementation_specific_data;
};

// Heuristic exceptions
exception HeuristicRollback {};
exception HeuristicCommit {};
exception HeuristicMixed {};
exception HeuristicHazard {};

// Other transaction-specific exceptions
exception SubtransactionsUnavailable {};
exception NotSubtransaction {};
```

```
exception Inactive {};
exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
exception SynchronizationUnavailable {};

// Current transaction
local interface Current : CORBA::Current {
    void begin()
        raises(SubtransactionsUnavailable);
    void commit(in boolean report_heuristics)
        raises(
            NoTransaction,
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback()
        raises(NoTransaction);
    void rollback_only()
        raises(NoTransaction);
    Status get_status();
    string get_transaction_name();
    void set_timeout(in unsigned long seconds);
    unsigned long get_timeout ();
    Control get_control();
    Control suspend();
    void resume(in Control which)
        raises(InvalidControl);
};

interface TransactionFactory {
    Control create(in unsigned long time_out);
    Control recreate(in PropagationContext ctx);
};

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

interface Terminator {
    void commit(in boolean report_heuristics)
        raises(
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback();
};
```

```
interface Coordinator {
    Status get_status();
    Status get_parent_status();
    Status get_top_level_status();

    boolean is_same_transaction(in Coordinator tc);
    boolean is_related_transaction(in Coordinator tc);
    boolean is_ancestor_transaction(in Coordinator tc);
    boolean is_descendant_transaction(in Coordinator tc);
    boolean is_top_level_transaction();

    unsigned long hash_transaction();
    unsigned long hash_top_level_tran();

    RecoveryCoordinator register_resource(in Resource r)
        raises(Inactive);

    void register_synchronization (in Synchronization sync)
        raises(Inactive, SynchronizationUnavailable);

    void register_subtran_aware(in SubtransactionAwareResource r)
        raises(Inactive, NotSubtransaction);

    void rollback_only()
        raises(Inactive);

    string get_transaction_name();

    Control create_subtransaction()
        raises(SubtransactionsUnavailable, Inactive);

    PropagationContext get_txcontext ()
        raises(Unavailable);
};

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

interface Resource {
    Vote prepare()
            raises(
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback()
```

```
            raises(
                HeuristicCommit,
                HeuristicMixed,
                HeuristicHazard
            );
        void commit()
            raises(
                NotPrepared,
                HeuristicRollback,
                HeuristicMixed,
                HeuristicHazard
            );
        void commit_one_phase()
            raises(
                HeuristicHazard
            );
        void forget();
    };

    // TransactionalObject has been deprecated
    // and replaced by the OTSPolicy component
    // Synchronization will use the OTSPolicy of ADAPTS
    // Inheritance from TransactionalObject is for backward compatability/

    interface Synchronization : TransactionalObject {
        void before_completion();
        void after_completion(in Status s);
    };

    interface SubtransactionAwareResource : Resource {
        void commit_subtransaction(in Coordinator parent);
        void rollback_subtransaction();
    };

    // TransactionalObject has been deprecated.
    interface TransactionalObject {
    };


    // TransactionPolicyType is deprecated and replaced
    // by InvocationPolicyType and OTSPolicyType
    // It is retained for backward compatibility.

    typedef unsigned short TransactionPolicyValue;

    const CORBA::PolicyType    TransactionPolicyType = 46;

    interface TransactionPolicy : CORBA::Policy {
        readonly attribute TransactionPolicyValue tpv;
    };
```

```
typedef unsigned short InvocationPolicyValue;

const InvocationPolicyValue EITHER = 0;
const InvocationPolicyValue SHARED = 1;
const InvocationPolicyValue UNSHARED =2;

typedef unsigned short OTSPolicyValue;

const OTSPolicyValue REQUIRES = 1;
const OTSPolicyValue FORBIDS =2;
const OTSPolicyValue ADAPTS =3;

typedef unsigned short NonTxTargetPolicyValue;

const NonTxTargetPolicyValue PREVENT = 0;
const NonTxTargetPolicyValue PERMIT = 1;

const CORBA::PolicyType INVOCATION_POLICY_TYPE = 55;

interface InvocationPolicy : CORBA::Policy {
    readonly attribute InvocationPolicyValue ipv;
};

const CORBA::PolicyType OTS_POLICY_TYPE = 56;

interface OTSPolicy : CORBA::Policy {
    readonly attribute OTSPolicyValue tpv;
};

const CORBA::PolicyType NON_TX_TARGET_POLICY_TYPE = 57;

interface NonTxTargetPolicy : CORBA::Policy {
    readonly attribute NonTxTargetPolicyValue tpv;
};

}; // End of CosTransactions Module
```

A.2 The CosTSPortability Module

```
module CosTSPortability { // PIDL
    typedef long ReqId;

    interface Sender {
        void sending_request(in ReqId id,
            out CosTransactions::PropagationContext ctx);
        void received_reply(in ReqId id,
            in CosTransactions::PropagationContext ctx,
            in CORBA::Environment env);
    };
```

```
    interface Receiver {
        void received_request(in ReqId id,
            in CosTransactions::PropagationContext ctx);
        void sending_reply(in ReqId id,
            out CosTransactions::PropagationContext ctx);
    };
};
```

A.3 The CosTSInteroperation Module

```
#include <orb.idl>
#include <IOP.idl>
module CosTSInteroperation {

    const IOP::ComponentId TAG_TRANSACTION_POLICY=26;

    struct TransactionPolicyComponent {
        CosTransactions::TransactionPolicyValue     tpv;
    };

    const IOP::ComponentId TAG_OTS_POLICY= 31;

    const IOP::ComponentId TAG_INV_POLICY= 32;
};
```

Relationship to TP Standards — B

Note – Editorial changes are in green.

This appendix discusses the relationship and possible interactions with the following related standards:

- X/Open TX interface
- X/Open XA interface
- OSI TP protocol
- LU 6.2 protocol
- ODMG standard

B.1 Support of X/Open TX Interface

B.1.1 Requirements

The X/Open DTP model[1] is now widely known and implemented.

Since the Transaction Service and the X/Open DTP models are interoperable, an application using transactional objects could use the TX interface, the X/Open-defined interface to delineate transactions, to interact with a Transaction Manager. (The Transaction Manager is the access point of the Transaction Service.)

1.See "Distributed Transaction Processing: The XA Specification, X/Open Document C193." X/Open Company Ltd., Reading, U.K., ISBN 1-85912-057-1.

B.1.2 TX Mappings

The correspondence between the TX interface primitives and the Transaction Service operations (**Current** interface) are as follows:

Table B-1 TX mappings

TX interface	*Current* interface
tx_open()	*no equivalent*
tx_close()	*no equivalent*
tx_begin()	Current::begin()
tx_rollback()	Current::rollback() or Current::rollback_only()
tx_commit()	Current::commit()
tx_set_commit_return()	report_heuristics parameter of Current::commit()
tx_set_transaction_control()	*no equivalent* *(chained transactions not supported)*
tx_set_transaction_timeout()	Current::set_timeout()
tx_info() - XID	Coordinator::get_txcontext() Current::get_name()[1]
tx_info() - COMMIT_RETURN	*no equivalent*
tx_info() - TRANSACTION_TIME_OUT	*no equivalent*
tx_info() - TRANSACTION_STATE	Current::get_status()

1. A printable string is output: not guaranteed to be the XID in all implementations.

tx_open

tx_open() provides a way to open, in a given execution environment, the Transaction Manager and the set of Resource Managers that are linked to it. Such an operation does not exist in the Transaction Service; such processing may be implicitly executed when the first operation of the Transaction Service is executed in the execution environment.

This processing is also related to a future Initialization Service.

tx_close

tx_close() provides a way to close, in a given execution environment, the Transaction Manager and the set of Resource Managers that are linked to it. Such an operation does not exist in the Transaction Service.

tx_begin

tx_begin() corresponds to **Current::begin()** or to **TransactionFactory::create()**.

tx_rollback

tx_rollback() corresponds to **Current::rollback()**, **Terminator::rollback()**, **Current::rollback_only()**, or **Coordinator::rollback_only()** . In TX, when a server calls **tx_rollback()**, the transaction may be rolled back or set as rollback only, as in the Transaction Service.

tx_commit and tx_set_commit_return

tx_commit() corresponds to **Current::commit()**. The Transaction Service operations have a parameter, **report_heuristics**, corresponding to the **commit_return** parameter of TX.

tx_set_transaction_control

tx_set_transaction_control() is used, in TX, to switch between unchained and chained mode; this function is not needed in the Transaction Service environment because it does not support chained transactions.

tx_set_transaction_timeout

tx_set_transaction_timeout() corresponds to **Current::set_timeout()** or **TransactionFactory::create()**.

tx_info

tx_info() returns information related to the current transaction. In the Transaction Service:

- the XID may be retrieved by **Coordinator::get_txcontext(**);
- the XID (in effect) may be retrieved by **Current::get_transaction_name()**;
- the transaction state may be retrieved by **Current::get_status()**;
- the commit return attribute is not needed because this attribute is given in the **commit()** operation;
- the timeout attribute cannot be obtained.

B.2 Support of X/Open Resource Managers

B.2.1 Introduction

In order to use a transactional system, such as a database system, with the Transaction Service, it is necessary to “hook” the transactions provided by this system and the distributed transactions managed by the Transaction Service.

With the Transaction Service, this is achieved by implementing **CosTransactions::Resource** objects — each resource represents a local transaction in the transactional system — and registering these Resource objects with the distributed transactions.

Since many systems provide a standard interface to their transactional capabilities — the XA interface — it is possible to implement **CosTransactions::Resource** objects on top of the XA interface, and provide an easy to use integration with the Transaction Service. The same integration (with the same interfaces and behavior) may also be provided through proprietary interfaces provided by a given Transaction Service implementation, without the creation and registration of **CosTransactions::Resource** objects. See Figure B-1.

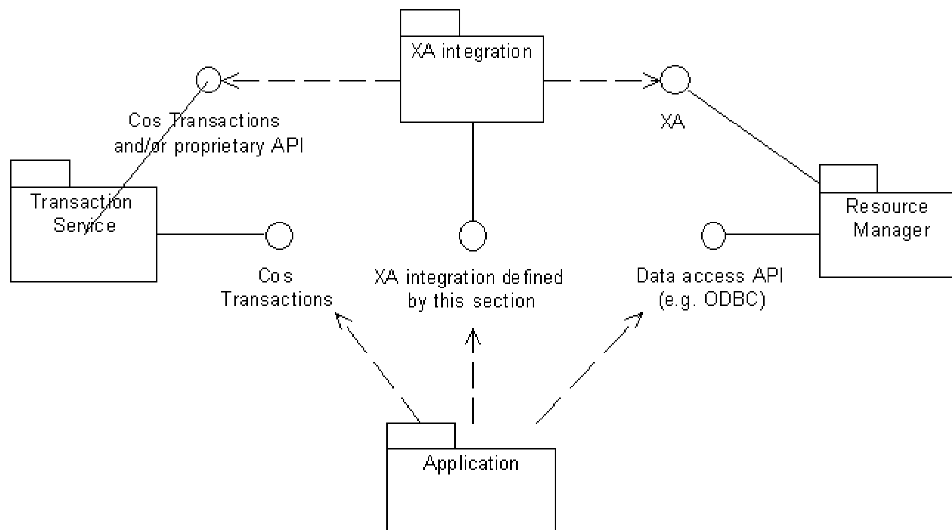

Figure B-1

The Java Transaction API Specification [JTA] defines the Java equivalent of the XA interface (**`javax.transaction.xa.XAResource`**) and a set of local Java interfaces that provide a "higher level" API to the Transaction Service (the interfaces are defined in the **`javax.transaction`** package). JTA also specifies the standard integration between the Transaction Service and Resource Managers that implement the Java **`XAResource`** interface.

For implementations in Java, JTA when available, is the preferred standard integration API between the Transaction Service and XA Resource Managers.

Note – JTA is the standard Transaction Service/XA Resource Manager integration API for implementations in Java compatible with the J2EE platform [J2EE].

This section specifies the interfaces for the standard integration between the Transaction Service and C XA resource managers[2]. Unlike JTA, this section does not define a higher level API to the Transaction Service: it relies directly on the Transaction Service types defined in the **CosTransactions** module.

This XA integration can be implemented using the standard Transaction Service interfaces; as a result, it may be provided by a Transaction Service vendor, a Resource Manager vendor, or any other third party. Likewise the integration with Resource Managers that implement the Java **XAResource** interface can be provided by a Transaction Service vendor, a Resource Manager vendor, or any other third party. A compliant Transaction Service implementation may, but does not need to, provide any or both of these standard integrations.

B.2.2 XA-compatible Transaction Service

An implementation of the Transaction Service is XA-compatible if it satisfies the following requirements:

- The Transaction Service does not restrict the availability of the **PropagationContext**: the operation **get_txcontext** on the **Coordinator** never raises Unavailable.
- The format of each **otid_t** value generated by the Transaction Service must correspond to the **XID** format, that is:
  - the **bqual_length** must be between 1 and 64
  - the **tid** length must be between **bqual_length** + 1 and **bqual_length** + 64
  - the gtrid (global transaction id) is provided by the first bytes in **tid**; the following **bqual_length** bytes correspond to the bqual (branch qualifier) part of the **XID**.
- Transactions in unrelated transaction families have distinct **otid_t** values.
- **otid_t** values generated by different XA-compatible Transaction Service implementations are always distinct.
  This is achieved by assigning **formatID**s to Transaction Service implementations: each XA-compatible Transaction Service implementation must use its own **formatID** value. **formatID** values other than 0 and -1[3] are assigned by the OMG. Allocation of **formatID**s may be requested by sending email to tag-request@omg.org.

2. This integration with C XA resource managers is briefly described in Sun's Java Transaction Service specification (http://java.sun.com/products/jts), "3.3 Support for pre-JTA Resource Managers"

3. 0 is reserved for OSI CCR naming. -1 means the XID is null.

The standard XA integration described below requires an XA-compatible implementation of the Transaction Service.

B.2.3 XA Overview

XA [XA] specifies a standard C API provided by transactional systems (called Resource Managers in the XA specification) that want to participate in distributed transactions managed by transaction managers developed by other vendors.

XA defines a set of C-function pointers, and a C-struct that holds these function pointers, **xa_switch_t**:

```
/*
 *  From Appendix A of the XA specification:
 */

struct xa_switch_t {
    char name[RMNAMESZ];           /* name of resource manager */
    long flags;                    /* resource manager specific options */
    long version;                  /* must be 0 */
    int (*xa_open_entry)           /* xa_open function pointer */
        (char *, int, long);
    int (*xa_close_entry)          /* xa_close function pointer */
        (char *, int, long);
    int (*xa_start_entry)          /* xa_start function pointer */
        (XID *, int, long);
    int (*xa_end_entry)            /* xa_end function pointer */
        (XID *, int, long);
    int (*xa_rollback_entry)       /* xa_rollback function pointer */
        (XID *, int, long);
    int (*xa_prepare_entry)        /* xa_prepare function pointer */
        (XID *, int, long);
    int (*xa_commit_entry)         /* xa_commit function pointer */
        (XID *, int, long);
    int (*xa_recover_entry)        /* xa_recover function pointer */
        (XID *, long, int, long);
    int (*xa_forget_entry)         /* xa_forget function pointer */
        (XID *, int, long);
    int (*xa_complete_entry)       /* xa_complete function pointer */
        (int *, int *, int, long);
};
```

Each XA-capable system must provide a global instance of **xa_switch_t**.

The function pointers provided by this **xa_switch_t** instances can be divided in four categories:

- functions to connect and disconnect to the XA resource manager:
  xa_open() and **xa_close()**
  The string passed to **xa_open()** typically contains connection information (e.g., a database name and a username and password).
- transaction completion functions:
  xa_prepare(), **xa_commit()**, **xa_rollback()**, **xa_forget()**
  They correspond to the **CosTransactions::Resource** operations.
- recovery functions
  xa_recover()
- functions used to start and end associations between connections and a transactions:
  xa_start(), **xa_end()**
  In order to use an XA connection to do some work within a distributed transaction, it is necessary to create an association between this connection and the distributed transaction.
  - **xa_start()** is used to create such an association;
  - **xa_end(TMSUSPEND)** suspends the association, without releasing the connection;
  - **xa_start(TMRESUME)** resumes a suspended association;
  - **xa_end(TMSUCCESS)** terminates an association with success, and
  - **xa_end(TMFAIL)** terminates an association and marks the transaction rollback-only.

xa_complete() is only used for asynchronous XA, an optional part of XA which is not supported by any popular XA implementation.

B.2.4 XA and Multi-Threading

In the XA specification, the scope of an XA connection is called thread of control: each thread-of-control can only use the connections that it has established (using **xa_open()**).

The XA specification maps thread-of-control to operating system process (see paragraph 2.2.8 in the XA specification), so one would expect that each thread in a process has access to all the XA connections established by this process. This is however not the case: most vendors implement the following:

- A thread-unsafe mode, in which the scope of each XA connection is the process (XA thread-of-control maps to process).
- A thread-safe mode, in which the scope of each XA connection is the thread by which is was created (XA thread-of-control maps to thread).

Sun's Java Transaction API provides a very different thread model: with JTA, every thread can use any connection (XAResource object).

The main drawback of tying connections and threads is flexibility since it prevents the application from managing connections independently of threads, which limits a lot the kind of connection pooling that can be implemented. Also, a CORBA server typically dispatches different requests to different threads: the thread of control equal thread model prevents the use of **`xa_end(TMSUSPEND)`** at the end of a request and **`xa_start(TMRESUME)`** at the beginning of the next request in the same transaction, since an association must be resumed by the thread of control by which it was suspended.

B.2.5 The Standard Integration with C XA Resource Managers

An implementation of the standard Transaction/Service XA integration implements the following three interfaces, defined in the **XA** module:

- **ResourceManager**
  A resource manager (logically) manages **CosTransactions::Resource** servants, or, using the XA vocabulary, transaction branches. **ResourceManager** is a distributed interface, which allows XA connections created in different ORB instances (and processes) to share the same **CosTransactions::Resource**/transaction branch.
- **CurrentConnection**
  A **CurrentConnection** is a local object that gives access to the XA connection associated with the current XA thread-of-control.
- **Connector**
  A local object used to create **ResourceManager** and **CurrentConnection** objects.

The XA module defines a fourth interface, **BeforeCompletionCallback**: it is implemented by applications that want to be notified before the completion of any transaction branch (**CosTransactions::Resource**) managed by a given **ResourceManager**.

B.2.6 CurrentConnection

The **CurrentConnection** local interface is defined in the **XA** module as follows:

```
typedef short ThreadModel;
const ThreadModel PROCESS = 0;
const ThreadModel THREAD  = 1;
local interface CurrentConnection
{
    void
    start(                              // xa_start(TMNOFLAGS) or xa_start(TMJOIN)
        in CosTransactions::Coordinator         tx,
        in CosTransactions::otid_t              otid
    );
```

```
    void
    suspend(                    // xa_end(TMSUSPEND)
        in CosTransactions::Coordinator  tx,
        in CosTransactions::otid_t       otid
    );
    void resume(                // xa_start(TMRESUME)
        in CosTransactions::Coordinator  tx,
        in CosTransactions::otid_t       otid
    );
    void end(                   // xa_end(TMSUCCESS) or xa_end(TMFAIL)
        in CosTransactions::Coordinator  tx,}
        in CosTransactions::otid_t       otid,
        in boolean                       success
    );
    ThreadModel thread_model();
    long        rmid();
};
```

start

start creates an association between an XA connection and a transaction branch.

In order to do some work within a distributed transaction with a given XA resource manager, the application needs to associate the resource manager's current connection with this transaction (or more precisely a transaction branch which represents this transaction in the resource manager), by calling **CurrentConnection::start**:

```
// C++
// assuming the OTS transaction is associated with the current thread

CosTransactions::Control_var control = tx_current->get_control();
CosTransactions::Coordinator_var tx = control->get_coordinator();
CosTransactions::PropagationContext_var ctx = tx->get_txcontext();
const CosTransactions::otid_t& otid = ctx->current.otid;
current_connection->start(tx, otid);
```

The first time **start** is called with a given **otid** on one of the **CurrentConnection** objects associated with a **ResourceManager**, the **ResourceManager** creates a transaction branch, creates a **CosTransactions::Resource** persistent object representing this transaction branch and registers this object with the given transaction coordinator. The **otid** parameter is transformed into an XID and passed to `xa_start()`, unaltered.

Note – A compliant implementation does not need to create and register a **CosTransaction::Resource** object, as long as the external behavior is the same.

---

suspend

suspend suspends an association between an XA connection and a transaction branch. It is merely a wrapper around **`xa_end(TMSUSPEND)`**. The **otid** parameter is transformed into an XID and passed to **`xa_end(TMSUSPEND)`**, unaltered. The **tx** parameter can be used to mark the transaction rollback-only when the implementation detects a fatal error.

resume

resume resumes a previously suspended association between an XA connection and a transaction branch. It is merely a wrapper around **`xa_start(TMRESUME`**). The **otid** parameter is transformed into an XID and passed to **`xa_start(TMRESUME)`**, unaltered. The **tx** parameter can be used to mark the transaction rollback-only when the implementation detects a fatal error.

end

ends the association between an XA connection and a transaction branch

Once the application has finished using a connection, it needs to end the association with the transaction branch, for two reasons:

- ending the association releases the connection, and makes it available for other transaction branches. (suspend has actually the same effect).
- as long as any connection is associated with a transaction branch, the transaction cannot be committed (even if the association is suspended). Some systems do not even allow to rollback a transaction branch while it is associated with any connection.

The **otid** parameter is transformed into an XID and passed to **`xa_end()`**, unaltered. The **tx** parameter can be used to mark the transaction rollback-only when the implementation detects a fatal error.

thread_model

thread_model returns the XA thread-of-control mapping used by this **CurrentConnection** object.

When the thread model is **PROCESS**, **`xa_open()`** is called by or before the first **start** call; **`xa_close()`** is called during shutdown. When the thread model is **THREAD**, each thread calls **`xa_open()`** the first time (or before the first time) this thread executes **CurrentConnection::start**; **`xa_close()`** is called when this thread exits.

rmid

rmid returns the rmid associated with this **CurrentConnection** object. The returned value is the same for any thread calling this operation.

The **otid_t** value cannot be simply extracted from the **tx** parameter because that would limit the users ability to optimize the calls under some circumstances (for example when using explicit propagation). It also allows to perform some operations (e.g. **end**) when the coordinator for a transaction is unreachable.

In addition, the user may wish to alter the branch qualifier of the **otid** to either share the same transaction branch between different processes (tightly-coupled model) or use different transaction branches in processes using the same resource manager within the same distributed transaction (loosely-coupled model).

Figure B-2 shows the components involved when the application creates a new association by calling **start** on a **CurrentConnection** object:

1. The application calls **start** on a **CurrentConnection** object.
2. The XA integration calls **`xa_start(TMNOFLAGS)`** to create a new transaction branch.
3. The XA integration creates a **Resource** object representing this branch, and registers this resource with the transaction coordinator.

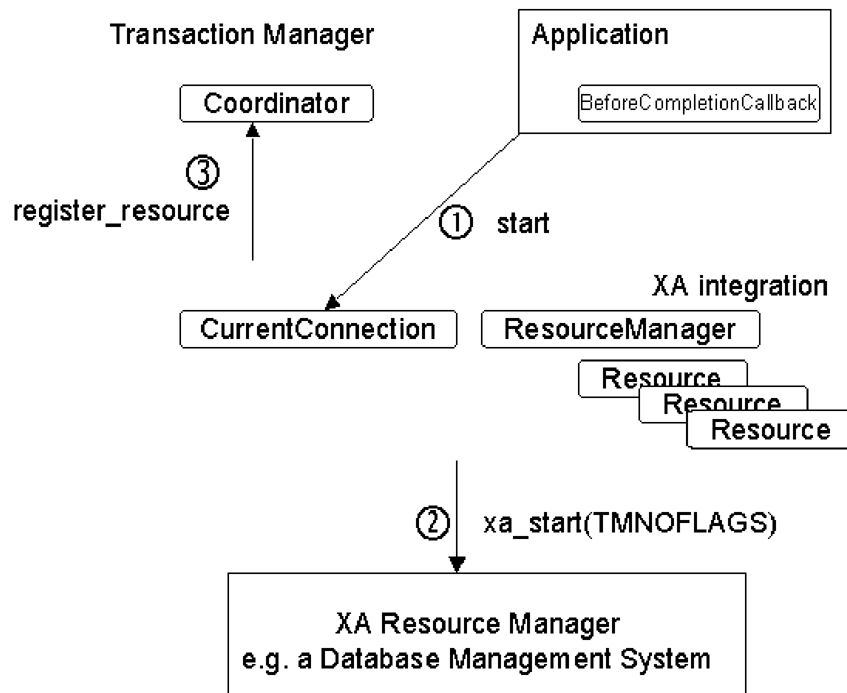

Figure B-2 Creating a New Association

B.2.7 Association State Diagram

Figure B-3 shows the state diagram of an association between a transaction and a XA connection. In this diagram all **start**, **suspend**, **resume**, and **end** calls are successful (they do not raise any exception).

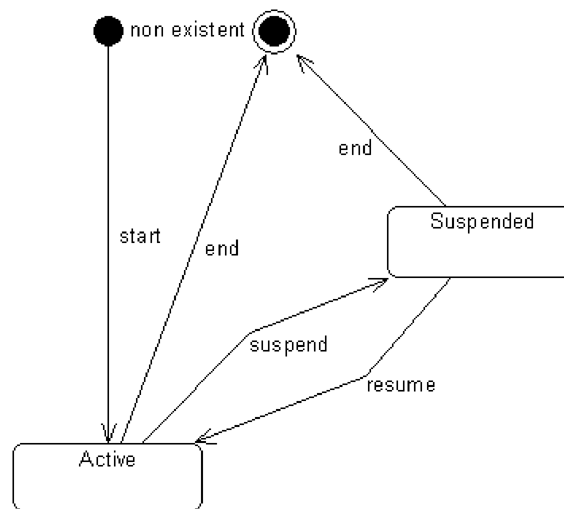

Figure B-3 Association State Diagram

When **start**, **suspend**, **resume** or **end** raises CORBA::INTERNAL with the minor code **2**, the new state is “non existent.”

When **resume**, **suspend** or **end** raises CORBA::TRANSACTION_ROLLEDBACK with the minor code **1**, the new state is “non existent.”

When **end** raises CORBA::TRANSACTION_ROLLEDBACK with the minor code **3**, the new state is “non existent.”

For every other exceptions raised by **start**, **suspend**, **resume** and **end**, there is no state transition.

Note – The PSS **TransactionalSession** interface has no **resume** operation: with PSS, when **start** is called on a suspended association, the association is resumed. Like PSS, JTA combines **start** and **resume** in a single method, `Transaction.enlistResource()`.

Because of the **CurrentConnection::resume** operation, an implementation of the **CurrentConnection** local interface does not need to maintain information about the state of the underlying XA connection(s).

ResourceManager

The **BeforeCompletionCallback** and the **ResourceManager** interfaces are defined in the **XA** module as follows:

```
interface BeforeCompletionCallback
{
    void
    before_completion(
        in CosTransactions::Coordinator     tx,
        in CosTransactions::otid_t          otid,
        in boolean                          success
    );
};

interface ResourceManager
{
    unsigned long
    register_before_completion_callback(in BeforeCompletionCallback bcc);

    void
    unregister_before_completion_callback(in unsigned long key);
};
```

A Resource Manager object manages transaction branches. An application can register any number (up to 2^31 -1) of **BeforeCompletionCallback** objects with a resource manager, to get notified each time a non-prepared transaction branch is about to be prepared, committed-in-one-phase or rolled back.

The only operation on **BeforeCompletionCallback**, **before_completion**, accepts the following parameters:

- **tx** - if available, **tx** is the transaction coordinator used in the call to **CurrentConnection::start** that created this transaction branch; else **tx** is nil.
- **otid** - the **otid_t** parameter used in the call to **CurrentConnection::start** that created this transaction branch.
- **success** - a boolean parameter which is TRUE when the non-preparated transaction branch is about to be prepared or committed in one phase, and FALSE when the transaction branch is about to be rolled back. If success is TRUE and **before_completion** raises any exception, the transaction branch is rolled back.

A typical use of a **BeforeCompletionCallback** object is to **end** a suspended association in a single-threaded server, as shown on Figure B-4.

An application uses the operation **register_before_completion_callback** to registers a **BeforeCompletionCallback** with a **ResourceManager**. **register_before_completion_callback** returns an unsigned long that the application uses to unregister a **BeforeCompletionCallback** object.

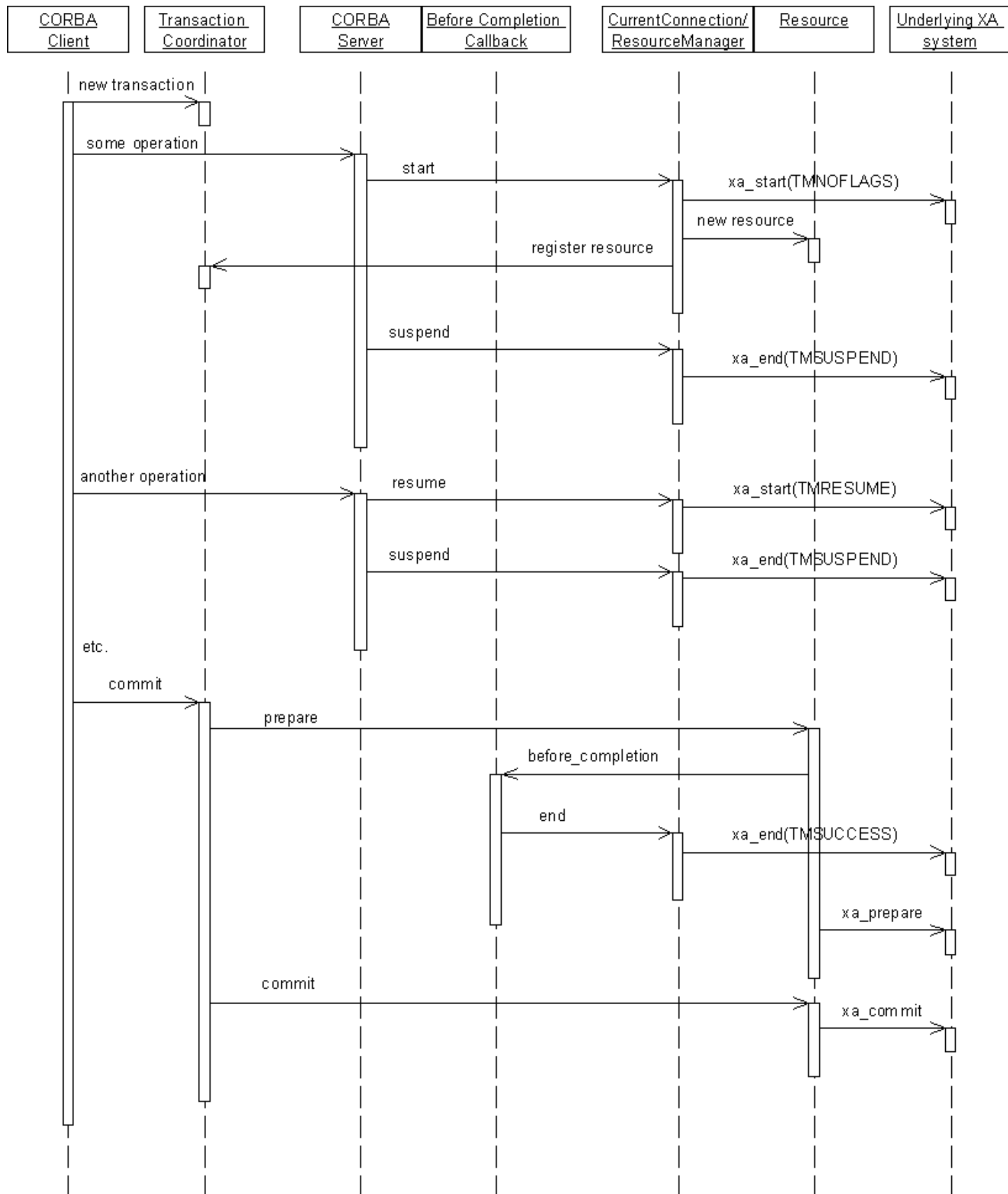

Figure B-4 Using a **BeforeCompletionCallback** to End a Suspended Association

Please note the following:

1. The primary advantage of **BeforeCompletionCallback** objects over **CosTransactions::Synchronization** objects is the number of CORBA requests per transaction: three for a synchronization (registration, before_completion, after_completion) versus only one for a **BeforeCompletionCallback** object.
2. When starting and ending an association for each request, there is no need for any **BeforeCompletionCallback** object.
3. The interfaces between the **CurrentConnection** objects, the **ResourceManager** object and the **Resource** objects (if there is any) are not specified. The diagram above illustrates a possible implementation. Other implementations are possible: for example a **Resource** object could register itself with the Transaction Coordinator in its constructor.

B.2.8 Connector

A connector local object is used to create **CurrentConnection** and **ResourceManager** objects. The connector itself is obtained by calling **resolve_initial_references()** on an ORB instance, with the `"XAConnector"` parameter.

The **Connector** local interface is defined in the **XA** module as follows:

```
native XASwitch;

local interface Connector
{
   ResourceManager
   create_resource_manager(
      in string               resource_manager_name,
      in XASwitch             xa_switch,
      in string               open_string,
      in string               close_string,
      in ThreadModel          thread_model,
      in boolean              automatic_association,
      in boolean              dynamic_registration_optimization,
      out CurrentConnection   current_connection
   );

   CurrentConnection
   connect_to_resource_manager(
      in ResourceManager      rm,
      in XASwitch             xa_switch,
      in string               open_string,
      in string               close_string,
      in ThreadModel          thread_model,
```

```
        in boolean              automatic_association,
        in boolean              dynamic_registration_optimization
    );
};
```

native XASwitch

In C and C++, the native type **XASwitch** maps to an `xa_switch_t`: an '**in**' **XASwitch** parameter is mapped to a `const xa_switch_t&` parameter in the corresponding C/C++ function.[4]

create_resource_manager

The **create_resource_manager** operation creates (or recreates) a **ResourceManager** object, and a **CurrentConnection** local object.

The **create_resource_manager** parameters are:

- (**in**) **resource_manager_name** - A string identifying the resource manager. This string may be used by the implementation to generate a POA name unique within some scope.
- (**in**) **xa_switch** - A const reference to the xa_switch_t global variable that gives access to the XA resource manager.
- (**in**) **open_string** - The XA resource manager open string .
- (**in**) **close_string** - The XA resource manager close string.
- (**in**) **thread_model** - The thread **model** used by this XA resource manager.
- (**in**) **automatic_association** - When TRUE, each time the ORB from which the **Connector** was retrieved receives a transactional request, the XA integration calls **start** on the **CurrentConnection** created by this operation, using the coordinator and current **otid** retrieved from the **PropagationContext**. When the processing of the transactional request is complete, the XA integration calls end on the **CurrentConnection** created by this operation, using the same coordinator and otid. The **success** parameter is TRUE when the operation completed without raising a standard exception, and FALSE otherwise. When FALSE, the XA integration is not involved during request processing.
- (**in**) **dynamic_registration_optimization** - When **dynamic_registration_optimization** is TRUE and the provided **xa_switch** supports dynamic registration, the XA integration may optimize the

4. Mappings for other languages that interface with C, for example Ada and Java, could be defined as well. For Java, it is expected that JTA-compliant `XAResource` implementations will be much more common than C `xa_switch_t` structs wrapped using JNI.

automatic_association processing described above by using XA dynamic registration: instead of calling **start** on the **CurrentConnection** when a transactional request is received, the XA integration relies on the **ax_reg()** call to create the association when needed. When **dynamic_registration_optimization** is FALSE, the **automatic_association** processing is as described above.

- (**out**) **current_connection** - A new **CurrentConnection** local object connected to the **ResourceManager** created (or recreated) by this operation.

The **ResourceManager** object reference returned by **create_resource_manager** is persistent.

connect_to_resource_manager

The **connector_to_resource_manager** operation creates a new **CurrentConnection** local object connected to an existing **ResourceManager**.

The **connect_to_resource_manager** parameters are:

- (**in**) **rm** - An object reference of the **ResourceManager** to connect to. This **ResourceManager** needs to be provided by the same XA integration implementation: XA integration A cannot create a **CurrentConnection** connected to a **ResourceManager** provided by XA integration B.
- (**in**) **xa_switch** - A const reference to the **xa_switch_t** global variable that gives access to the XA resource manager.
- (**in**) **open_string** - The XA resource manager open string .
- (**in**) **close_string** - The XA resource manager close string.
- (**in**) **thread_model** - The thread model used by this XA resource manager.
- (**in**) **automatic_association** - When TRUE, each time the ORB from which the **Connector** was retrieved receives a transactional request, the XA integration calls **start** on the **CurrentConnection** created by this operation, using the coordinator and current otid retrieved from the **PropagationContext**. When the processing of the transactional request is complete, the XA integration calls **end** on the **CurrentConnection** created by this operation, using the same coordinator and otid. The success parameter is TRUE when the operation completed without raising a standard exception, and FALSE otherwise. When FALSE, the XA integration is not involved during request processing.
- (**in**) **dynamic_registration_optimization** - When **dynamic_registration_optimization** is TRUE and the provided **xa_switch** supports dynamic registration, the XA integration may optimize the **automatic_association** processing described above by using XA dynamic registration: instead of calling start on the **CurrentConnection** when a transactional request is received, the XA integration relies on the **ax_reg()** call to create the association when needed. When **dynamic_registration_optimization** is FALSE, the **automatic_association** processing is as described above.

B.2.9 Exceptions and Minor Codes

All the operations on the interfaces defined in the XA module can only raise standard exceptions. For portability, the table below defines which exceptions are raised in some circumstances, and the minor code of these exceptions. The mappings are summarized in the following table:

Table B-2 Exceptions and Minor Codes

Exception	Minor Code	Raised By/When
BAD_PARAM	33	Connector::create_resource_manager, Connector::connector_to_resource_manager, CurrentConnection::start, CurrentConnection::suspend, CurrentConnection::resume, CurrentConnection::end when an xa_ call returns **XAER_INVAL**
BAD_INV_ORDER	19	CurrentConnection::start when an **xa_start()** call returns **XAER_OUTSIDE**
BAD_INV_ORDER	20	CurrentConnection::start, CurrentConnection::suspend, CurrentConnection::resume, CurrentConnection::end when an xa_ call returns **XAER_PROTO**
INTERNAL	1	Connector::create_resource_ manager, Connector::connector_to_ resource_manager CurrentConnection::start, CurrentConnection::suspend, CurrentConnection::resume, CurrentConnection::end when an xa_ call returns **XAER_RMERR**
INTERNAL	2	CurrentConnection::start, CurrentConnection::suspend, CurrentConnection::resume, CurrentConnection::end when an xa_ call returns **XAER_RMFAIL**

Table B-2 Exceptions and Minor Codes

Exception	Minor Code	Raised By/When
TRANSACTION_ROLLEDBACK	1	**CurrentConnection::start, CurrentConnection::suspend, CurrentConnection::resume, CurrentConnection::end** when an xa_ call returns an **XAER_RB** error code
TRANSACTION_ROLLEDBACK	2	**CurrentConnection::start, CurrentConnection::suspend, CurrentConnection::resume, CurrentConnection::end** when an xa_ call returns **XAER_NOTA**
TRANSACTION_ROLLEDBACK	3	**CurrentConnection::end** In some circumstances, the XA integration may defer the rollback of a transaction branch until an association with this branch is ended. When this occurs and **end** is called with success set to TRUE, **end** raises **TRANSACTION_ROLLEDBACK** with the 3 minor code.

B.2.10 Comparison with the Java Transaction API (JTA)

The following informational tables show the correspondance between the Transaction/XA integration IDL interfaces and some JTA interfaces, the XA C function pointers and JTA **XAResource** methods, and between the OMG Transaction Service IDL interfaces and JTA's transaction manager interfaces.

Table B-3 Comparing the OTS/XA integration with JTA

OTS/XA integration	JTA
CurrentConnection::start(Coordinator, otid_t)	`Transaction.enlistResource(XAResource)`
CurrentConnection::suspend (Coordinator, otid_t)	`Transaction.delistResource(XAResource, TMSUSPEND)`
CurrentConnection::resume (Coordinator, otid_t)	`Transaction.enlistResource(XAResource)`
CurrentConnection::end (Coordinator, otid_t, boolean)	`Transaction.delistResource(XAResource, TMSUCCESS)` or `Transaction.delistResource(XAResource, TMFAIL)`
ResourceManager	no equivalent
BeforeCompletionCallback	no equivalent

Table B-3 Comparing the OTS/XA integration with JTA

OTS/XA integration	JTA
CurrentConnection	no equivalent, although the notion of XA connection is the same in the OTS/XA integration and in JTA
Connector	no equivalent
Automatic association	no equivalent

Table B-4 Comparing XA with JTA

XA	JTA
A `xa_switch_t` object	A transactional resource factory
An opened rmid	A `XAResource` object
`xa_open_entry(char *, int, long)`	no equivalent; JTA does not standardize an API to open or close XA connections
`xa_close_entry(char *, int, long)`	no equivalent
`xa_start_entry(XID *, int, long)`	`XAResource.start(Xid, int)`
`xa_end_entry(XID *, int, long)`	`XAResource.end(Xid, int)`
`xa_rollback_entry(XID *, int, long)`	`XAResource.rollback(Xid, int)`
`xa_prepare(XID *, int, long)`	`XAResource.prepare(Xid, int)`
`xa_commit(XID *, int, long)`	`XAResource.commit(Xid, int)`
`xa_recover(XID *, long, int, long)`	`XAResource.recover(int)`
`xa_forget(XID *, int, long)`	`XAResource.forget(Xid, int)`
`xa_complete(int *, int *, int, long)`	no equivalent
no equivalent	`XAResource.getTransactionTimeout()`
no equivalent	`XAResource.isSameRM(XAResource)`
no equivalent	`XAResource.setTransactionTimeout(int)`

Table B-5 Comparing the Transaction Service with JTA

Transaction Service	JTA
Current	`TransactionManager, UserTransaction`
Synchronization	`Synchronization`
Control/Coordinator/Terminator	`Transaction`
Current::begin	`TransactionManager.begin(),UserTransaction.begin()`
Current::commit	`TransactionManager.commit(),UserTransaction.commit()`

Table B-5 Comparing the Transaction Service with JTA

Transaction Service	JTA
Current::rollback	`TransactionManager.rollback()`, `UserTransaction.rollback()`
Current::rollback_only	`TransactionManager.setRollbackOnly()`, `UserTransaction.setRollbackOnly()`
Current::get_status	`TransactionManager.getStatus()`, `UserTransaction.getStatus()`
Current::get_transaction_name	`TransactionManager.toString()`, `UserTransaction.toString()`
Current::set_timeout	`TransactionManager.setTransactionTimeout()`
Current::get_control	`TransactionManager.getTransaction()`
Current::suspend	`TransactionManager.suspend()`
Current::resume	`TransactionManager.resume()`
Coordinator::get_status	`Transaction.getStatus()`
Coordinator::is_same_transaction	`Transaction.equals()`
Coordinator::hash_transaction	`Transaction.hashCode()`
Coordinator::register_synchronization	`Transaction.registerSynchronization()`
Coordinator::rollback_only	`Transaction.setRollbackOnly()`
Terminator::commit()	`Transaction.commit()`
Terminator::rollback	`Transaction.rollback()`
Other **Coordinator** operations	no equivalent

B.2.11 XA Module

```
#ifndef _XA_IDL_
#define _XA_IDL_
#include <CosTransactions.idl>
#pragma prefix "omg.org"

module XA
{
   native XASwitch;

   typedef short ThreadModel;
   const ThreadModel PROCESS = 0;
   const ThreadModel THREAD  = 1;

   local interface CurrentConnection
   {
      void
      start(                                  // xa_start(TMNOFLAGS) or xa_start(TMJOIN)
         in CosTransactions::Coordinator      tx,
         in CosTransactions::otid_t           otid
      );
```

```
    void
    suspend(                              // xa_end(TMSUSPEND)
        in CosTransactions::Coordinator        tx,
        in CosTransactions::otid_t             otid
    );

    void resume(                          // xa_start(TMRESUME)
        in CosTransactions::Coordinator        tx,
        in CosTransactions::otid_t             otid
    );

    void end(                             // xa_end(TMSUCCESS) or xa_end(TMFAIL)
        in CosTransactions::Coordinator        tx,
        in CosTransactions::otid_t             otid,
        in boolean                             success
    );

    ThreadModel thread_model();
    long            rmid();

};

interface BeforeCompletionCallback
{
    void
    before_completion(
        in CosTransactions::Coordinator        tx,
        in CosTransactions::otid_t             otid,
        in boolean                             success
    );
};

interface ResourceManager
{
    unsigned long
    register_before_completion_callback(in BeforeCompletionCallback bcc);

    void
    unregister_before_completion_callback(in unsigned long key);
};


local interface Connector
{
    ResourceManager
    create_resource_manager(
        in string               resource_manager_name,
        in XASwitch             xa_switch,
        in string               open_string,
        in string               close_string,
        in ThreadModel          thread_model,
        in boolean              automatic_association,
        in boolean              dynamic_registration_optimization,
        out CurrentConnection   current_connection
```

```
        );

        CurrentConnection
        connect_to_resource_manager(
            in ResourceManager        rm,
            in XASwitch               xa_switch,
            in string                 open_string,
            in string                 close_string,
            in ThreadModel            thread_model,
            in boolean                automatic_association,
            in boolean                dynamic_registration_optimization
        );
    };
};

#endif  /*!_XA_IDL_*/
```

B.2.12 References

[J2EE] Java 2 Platform Enterprise Edition (J2EE), Platform specification: http://java.sun.com/j2ee

[JTA] Java Transaction API (JTA): http://java.sun.com/jta

[XA] Open Group Technical Standard, Distributed TP: The XA Specification, February 1991, ISBN: 1 872630 24 3

B.3 Interoperation with Transactional Protocols

B.3.13 Transactional Protocols

A CORBA application may sometimes need to interoperate with one or more applications using one of the de-facto standard transactional protocol: OSI TP and SNA LU 6.2. In this case, the Transaction Service must be able to import or export transactions using one of these protocols.

Export is the ability to relate a transaction of the Transaction Service to a transaction of a foreign transactional protocol. Importing means relating a Transaction Service transaction to a transaction started on a remote application and propagated via the foreign transactional protocol.

Since the model used by the Transaction Service is similar to the model of OSI TP and the X/Open DTP model, the interactions with OSI TP are straightforward. Since OSI TP is a compatible superset of SNA LU 6.2, a mapping to SNA communications is easily accomplished.

To interoperate, a mapping should be defined for the two-phase commit, rollback, and recovery mechanisms, and for the transaction identifiers.

Notice that neither OSI TP nor SNA LU 6.2 supports nested transactions.

B.3.14 OSI TP Interoperability

OSI TP [ISO92] is the transactional protocol defined by ISO. It has been selected by X/Open to allow the distribution of transactions by one of the communication interfaces: remote procedure call[5], client-server [6] or peer-to-peer (CPI-C Level-2 API [CIW93]).

The Transaction Service supports only unchained transactions. The use of dialogues using the Chained Transactions functional unit is possible only if restrictive rules are defined. These rules are not described in this document.

OSI TP Transaction Identifiers

In OSI TP, loosely-coupled transactions are supported and every node of the transaction tree possesses a transaction branch identifier which is composed of the transaction identifier (or atomic action identifier) and a branch identifier (the branch identifier being null for the root node of the transaction tree). Both the transaction identifier and the branch identifier contains an AE-Title (Application Entity Title) and a suffix that make it unique within a certain scope.

The format of the standard X/Open XID is compatible with the OSI TP identifiers, the gtrid corresponding to the atomic action identifier and the bqual corresponding to the branch identifier.

Incoming OSI TP Communications (Imported Transactions)

The Transaction Service is a subordinate in an OSI TP transaction tree and interacts with its superior by regular PDUs as defined by the OSI TP protocol. The Transaction Service introduces the transaction identifier received on the OSI TP dialogue using the TransactionFactory::recreate operation.

The Transaction Service maps the OSI TP commitment, rollback and recovery procedures to the Transaction Service commitment procedure as follows:

- The Transaction Service, upon reception of an OSI TP Prepare message, will enter the first phase of commitment procedure.
- When it enters the prepared state for the transaction, the Transaction Service will trigger the sending of an OSI TP Ready message to its superior. (It may trigger a Recover (Ready) message when normal communications are broken with the superior).

---

5. See "Distributed Transaction Processing: The TxRPC Specification, X/Open Document P305." X/Open Company Ltd., Reading, U.K.

6. See "Distributed Transaction Processing: The XATMI Specification, X/Open Document P306." X/Open Company Ltd., Reading, U.K.

- The Transaction Service, upon reception of an OSI TP Commit message, enters the second phase of commitment procedure. (It may receive a Recover (Commit) when normal communications are broken with the superior.)
- The Transaction Service, upon reception of an OSI TP Rollback message (it may be a Recover (Unknown) when normal communications are broken with the superior or any other rollback-initiating condition) will enter its rollback procedure (unless a rollback is already in progress).
- The Transaction Service, upon reception of the last rollback reply, will trigger the sending of a Rollback Response/Confirm message to its superior.

Outgoing OSI TP Communications (Exported Transactions)

The Transaction Service behaves as a superior in an OSI TP transaction tree and interacts with its subordinates by regular PDUs as defined by the OSI TP protocol.

The Transaction Service will map the OSI TP commitment procedure as follows:

- The Transaction Service, during the first phase of commitment procedure will invoke an OSI TP Prepare message to all its subordinates.
- Upon reception of an OSI TP Ready message, the Transaction Service will process this message as a successful reply to prepare.
- The Transaction Service, upon entering the second phase of the commitment procedure will send an OSI TP Commit message (it may be a Recover (Commit) when normal communications are broken with the subordinate) to all subordinates.
- The Transaction Service, upon reception of an OSI TP Rollback message (it may be any other rollback-initiating condition) will enter its rollback procedure (unless a rollback is already in progress).
- The Transaction Service, upon reception of the last Rollback Response/Confirm message from its subordinates, will process this message as a reply to a rollback operation and determine the heuristic situation.

SNA LU 6.2 Interoperability

SNA LU 6.2 ([SNA88a], [SNA88b]) is a transactional protocol defined by IBM. It is widely used for transaction distribution. The standard interface to access LU 6.2 communications is CPI-C (Common Programming Interface for Communications) defined by IBM in the context of SAA [CPIC93] and currently being evolved by the CPI-C Implementers' Workshop to become CPI-C level 2, a modern interface usable for LU 6.2 and OSI TP communications [CIW93].

LU 6.2 supports only chained transactions but, at a given node, a transaction is started only when resources have been involved in the transaction. LU 6.2 can be used for a portion of an “unchained” transaction tree if the LU 6.2 conversations are ended after each transaction by any node that has both LU 6.2 conversations and dialogues of an unchained transaction.

LU 6.2 Transaction Identifiers

SNA LU 6.2 also supports loosely-coupled transactions and uses a specific format for transaction identifiers: the Logical Unit of Work (LUWID) corresponds to the OSI Transaction Identifier. The LUWID is composed of:

- The Fully Qualified Logical Unit Name, which is composed of up to 17 bytes, is unique in an SNA network or a set of interconnected SNA networks.
- An instance number which is unique at the LU that create the transaction.
- The sequence number that is incremented whenever the transaction is committed.

The Conversation Correlator corresponds to the OSI TP Branch Identifier; it is a string of 1 to 8 bytes which are unique within the context of the LU having established the conversation and is meaningful when combined with the Fully Qualified LU Name of this Logical Unit.

Incoming LU 6.2 Communications

The LU 6.2 two-phase commit protocol is different from the OSI TP protocol: the system sending a Prepare message has to perform logging and is responsible for recovery. LU 6.2 does also support features like last-agent optimization, read-only and allows any node in the transaction tree to request commitment.

The Transaction Service is a subordinate in an LU 6.2 transaction tree and interacts with its superior using SNA requests and responses as defined by the LU 6.2 protocol. The Transaction Service maps the LUWID corresponding to the incoming conversation to an OMG **otid_t** and issues TransactionFactory::recreate to import the transaction.

The Transaction Service maps the LU 6.2 commitment, rollback and recovery procedures to the Transaction Service commitment procedure as follows:

- The Transaction Service, upon reception of an LU 6.2 Prepare message will enter the first phase of commitment procedure.
- The Transaction Service, upon entering the prepared state for the transaction, the Transaction Service will trigger the sending of a Request Commit message to is superior.
- The Transaction Service, upon reception of an LU 6.2 Committed message (it may be a Compare States (Committed) when normal communications are broken with the superior) will enter the second phase of commitment procedure.
- The Transaction Service, upon leaving the decided commit state, will trigger the sending of a Forget message to is superior (it may be a Reset when normal communications are broken with the superior).

Due to the two-phase commit difference, the Transaction Service will never send the equivalent of the Recover(Ready) unless prompted by the superior.

The last-agent and read-only features may also be supported by the Transaction Service.

Outgoing LU 6.2 Communications

The Transaction Service has to log when the Prepare message is sent and, in case of communication failure or restart of the Transaction Service, a recovery is needed.

ODMG Standard

ODMG-93 is a standard defined by ODMG (Object Database Management Group) describing portable interface to access Object Database Management Systems (ODBMS).

Since it is likely that, in the future, many objects involved in transactions will be handled by an ODBMS, this standard has a strong relationship with the Transaction Service.

B.4 ODMG Model

The ODMG model defines optional transactions and supports the nested transaction concept. The ODMG model does not cover the integration of ODBMS with an external Transaction Service, allowing other resources and communications to be involved in a transaction. No two-phase commit or recovery protocol is described.

A transaction object must be created. The transactional operations are:

- Begin (or start) to begin a transaction (or a subtransaction).
- Commit to request commitment of a transaction.
- Abort to rollback a transaction.
- Checkpoint to commit the transaction but keep the locks. This feature is not supported by the current version of the Transaction Service.
- abort_to_top_level to request rollback of a nested transaction family. The Transaction Service does not directly support this feature but does provide means to perform this functionality by resuming the context of the top-level transaction and then requesting rollback.

If the transaction object is destroyed, the transaction is rolled back.

B.4.1 Integration of ODMG ODBMSs with the Transaction Service

Since ODMG-93 does not define any way to integrate an ODBMS into an existing transaction, the integration is difficult unless the ODBMS supports the XA interface, in which case the section on XA-compliant RM is applicable.

In the future, it is anticipated that ODBMS will implement the Transaction Service-defined interfaces and be considered as a recoverable server.

A possibility is to use, at a root node, an ODBMS as a last resource and, after all subordinates are prepared, to request a one-phase commitment to the ODBMS. If the outcome for the ODBMS is commit, the transaction will be committed, if it is rollback,

the transaction will be rolled back. The mechanism may work if it is possible to determine, after a crash, whether the ODBMS committed or rolled back; this may be done at application level.

Conformance Requirements C

Note – Text in red is from the Persistent State Service (PSS) specification.

A conformant Transaction Service implements all the features that are not described as optional in this specification.

However, to satisfy the lite conformance level, an implementation does not need to support the registration of more than one resource per transaction, and it does not need to support distribution (the flow of transactions from ORB context to ORB context).

To satisfy the lite-distributed conformance level, an implementation must support distribution.

To satisfy the full conformance level, an implementation must support distribution and the registration of multiple resources per transaction.

Glossary

2PC — See *Two-phase commit.*

Abort — See *Rollback*

Active — The state of a transaction when processing is in progress and *completion* of the transaction has not yet commenced.

Atomicity — A transaction property that ensures that if work is interrupted by failure, any partially completed results will be undone. A transaction whose work completes is said to commit. A transaction whose work is completely undone is said to rollback (abort).

Begin — An operation on the Transaction Service which establishes the initial boundary of a transaction.

Commit — Commit has two definitions as follows:

An operation in the *Current* and *Terminator* interfaces that a program uses to request that the current transaction terminate normally and that the effects of that transaction be made permanent.

An operation in the *Resource* interface which causes the effects a transaction to be made permanent.

Commit coordinator — In a two-phase commit protocol, the program that collects the vote from the participants.

Commit participant — In a two-phase commit protocol, the program that returns a vote on the completion of a transaction.

Committed — The property of a transaction or a transactional object, when it has successfully performed the commit protocol. See also *in-doubt*, *active*, and *rolled back*.

Completion The processing required (either by *commit* or *rollback*) to obtain the durable outcome of a transaction.

Coordinator A coordinator involves *Resource* objects in a transaction when they are registered. A coordinator is responsible for driving the two-phase commit protocol. See also *Commit coordinator* and *Commit participant*.

Consistency A property of a transaction that ensures that the transaction's actions, taken as a group, do not violate any of the integrity constraints associated with the state of its associated objects. This requires that the application program be implemented correctly: the Transaction Service provides the functionality to support application data consistency.

Decided commit state A root coordinator enters the decided commit state when it has written a log-commit record; a subordinate coordinator or resource is in the decided commit state when it has received the commit instruction from its superior; in the latter case, a log-commit record may be written but this is not essential.

Decided rollback state A coordinator or resource enters the decided rollback state when it decides to rollback the transaction or has received a signal to do so.

Direct context management An application manipulates the *Control* object and the other objects associated with the transaction. See also *Indirect context management.*

Durability A transaction property that ensures the results of a successfully completed transaction will never be lost, except in the event of catastrophe. It is generally implemented by a combination of persistent storage and a logging service that provides a backup copy of permanent changes.

Execution environment An implementation-dependent factor that may determine the outcome of certain operations on the Transaction Service. Typically the execution environment is the scope within which shared state is managed.

Flat Transaction A transaction that has no subtransactions—and that cannot have subtransactions.

Forgotten "state" This is not really a transaction state at all, because there is no memory of the transaction: it has either completed or rolled back and all records on permanent storage have been deleted.

Heuristic Commit or Rollback To unilaterally make the commit or rollback decision about *in-doubt* transactions when the coordinator fails or contact with the coordinator fails.

Indirect context management An application uses the *Current* object, provided by the Transaction Service, to associate the transaction context with the application thread of control. See also *Direct context management.*

In-doubt The state of a transaction if it is controlled by a transaction manager that can not be contacted, so the commit decision is in doubt. See also *active*, *committed*, *rolled back*.

Interposition Adding a sequence of one or more *subordinate coordinators* between a *root coordinator* and its participants.

Isolation A transaction property that allows concurrent execution, but the results will be the same as if execution was serialized. Isolation ensures that concurrently executing transactions cannot observe inconsistencies in shared data.

Lock service Called the Concurrency Control Service, it is an Object Service used by resources to control access to shared objects by concurrently executing methods.

Log-ready record (and contents) for an intermediate coordinator a log-ready record contains identification of the (superior) coordinator and of *Resource* objects (including subordinate coordinators) registered with the coordinator which replied VoteCommit (i.e., it excludes registered objects that replied VoteReadOnly); for a *Resource* object a log-ready record includes identification of the coordinator with which it is registered.

Log-commit record (and contents) A log-commit record contains identification of all registered *Resource* objects that replied VoteCommit.

Log-heuristic record This contains a record of a heuristic decision either HeuristicCommit or HeuristicRollback.

Log-damage record This contains a record of heuristic damage i.e. where it is known that a heuristic decision conflicted with the decided outcome (HeuristicMixed) or where there is a risk that a heuristic decision conflicted with the decided outcome (HeuristicHazard).

Log service A service used by resource managers for recording recovery information and the Transaction Service for recording transaction state durably.

Nested transaction A transaction that either has subtransaction or is a subtransaction on some other transaction.

Participant See *Commit participant.*

Persistent storage Generally speaking, a synonym for *Stable storage*. In the context of the OMA, the Persistent Object Service (POS) provides an object representation of stable storage.

Prepared The state that a transaction is in when phase one of a two-phase commit has completed.

Presumed rollback An optimization of the two-phase commit protocol that results in more efficient performance as the *root coordinator* does not need to log anything before the commit decision and the *Participants* (i.e., *Resource* objects) do not need to log anything before they prepare. So called because, at restart, if no record of the transaction is found, it is safe to assume the transaction rolled back.

Propagation A function of the Transaction Service that allows the *Transaction context* of a client to be associated with a transactional operation on a server object. The Transaction Service supports both implicit and explicit propagation of transaction context.

Recoverable Object An object whose data is affected by committing or rolling back a transaction.

Recoverable Server A transactional object with recoverable state that registers a *Resource* (not necessarily itself) with a *Coordinator* to participate in transaction completion.

Recovery Service A service used by resource managers for restoring the state of objects to a prior state of consistency.

Resource An object in the Transaction Service that is registered for involvement in two-phase commit—*2PC*. Corresponds to a *Resource Manager.*

Resource Manager An X/Open term for a component which manages the integrity of the state of a set of related resources.

Rollback Rollback (also known as *Abort*) has two definitions, as follows:

An operation in the *Current* and *Terminator* interfaces used to indicate that the current transaction has terminated abnormally and its effects should be discarded.

An operation in the *Resource* interface which causes all state changes in the transaction to be undone.

Rolled Back The property of a transaction or a transactional object when it has discarded all changes made in the current transaction. See also *in-doubt*, *active*, and *committed*.

Root Coordinator The first coordinator in a sequence of coordinators where there is interposition. The coordinator associated with the transaction originator.

Security Service An object service which provides identifications of users (authentication), controls access to resources (authorization), and provides auditing of resource access.

Stable storage Storage not likely to be damaged as the result of node failure.

Sub-coordinator See *Subordinate Coordinator.*

Subordinate Coordinator A coordinator subordinate to the *root coordinator* when *interposition* has been performed. A subordinate coordinator appears as a *Resource* object to its superior. Also known as a *Sub-coordinator.*

Synchronization An object in the Transaction Service which controls the transfer of persistent object state data so it can be made durable by its associated resource.

Thread The entity that is currently in control of the processor.

Thread Service — A service which enables methods to be executed concurrently by the same process. Where two or more methods can execute concurrently each method is associated with its own thread of control.

TP monitor — A system component that accepts input work requests and associates resources with the programs that act upon these requests to provide a run-time environment for program execution.

Transaction — A collection of operations on the physical and abstract application state.

Transactional client — An arbitrary program that can invoke operations of many transactional objects in a single transaction. Not necessarily the *Transaction originator.*

Transaction Context — The transaction information associated with a specific thread. See *Propagation*.

Transactional operation — An operation on an object that participates in the propagation of the current transaction.

Transaction originator — An arbitrary program—typically, a transactional client, but not necessarily an object—that begins a transaction.

Transaction Manager — A system component that implements the protocol engine for 2-phase commit protocol. See also *Transaction Service*.

Transactional object — An object whose operations are affected by being invoked within the scope of a transaction.

Transactional server — A collection of one or more objects whose behavior is affected by the transaction, but has no recoverable state of its own.

Transaction Service — An Object Service that implements the protocols required to guarantee the ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions. See also *Transaction Manager.*

TSPortability — An interface of the Transaction Service which allows it to track transactional operations and propagate transaction context to another Transaction Service implementation.

Two-Phase commit — A transaction manager protocol for ensuring that all changes to recoverable resources occur atomically and furthermore, the failure of any resource to complete will cause all other resource to undo changes. Also called *2PC*.

Index

A
abort
  see rollback
atomicity 2-40, 2-43, 2-48
  glossary definition 1

C
callback interface
  described x
common facilities vi
compound objec tix
concepts of viii
Control interfa c e2-6
control object 2-6, 2-13, 2-55
Coordinator interface 2-8
  create_subtransaction operation 2-13
  get_parent_status operation 2-10
  get_status operation 2-9
  get_top_level_status operation 2-10
  get_transaction_name operation 2-13
  hash_top_level_tran operatio n2-11
  hash_transaction operatio n2-11
  is_ancestor_transaction operation 2-11
  is_descendant_transacation operatio n2-11
  is_related_transaction operatio n2-11
  is_same_transaction operatio n2-10
  is_top_level_transaction operatio n2-11
  register_resource operation 2-11
  register_subtran_aware operation 2-12
  rollback_only operation 2-12
coordinator object 2-15, 2-17, 2-32, 2-33, 2-44, 2-55
  glossary definition 2
CORBA viii
  contributors xi
  documentation set vii
CosTransactions module
  datatypes defined b y1-14
  OMG IDL A-1
CosTSInteroperation module
  PIDL 2-56, A-4, A-5
Current interfa c e2-32

E
event channel ix, x
EventChannel interfac eix
exceptions
  described xi

G
global identifie rx

I
interface inheritance.see subtyping

O
Object Management Group v
  address of viii
object model vii
object request broker vi, vii
object service
  context vi
  specification defined vii
ODMG-93 protocol B-11
  integration with transaction service B-12
OMG IDL vii, ix
OSI TP protocol B-8
  exported transactions B-9
  imported transactions B-9
  transaction identifiers B-8

P
propagation 2-29, 2-32, 2-35, 2-54, 2-58, 2-59
  glossary definition 4
PullSupplier interface ix
PushConsumer interface ix

Q
quality of service ix

R
recoverable object 1-5
  and nested transactio n s2-17
recoverable server 1-6, 2-33
  glossary definition 4
  implementing 2-30
RecoveryCoordinator interface 2-13
  replay_completion operation 2-13
reference model vi
Resource interface 2-14
  commit operation 2-15
  commit_one_phase operation 2-16
  forget operation 2-16
  prepare operation 2-14
  rollback operatio n2-15
resource manager 1-9, 2-63, B-6
  mappings to B-4
resource object
  defined 1-5
RM
  see resource manager
rollback
  glossary definition 4

S
SNA LU protocol B-8, B-10
  incoming communication B-10
  outgoing communication B-11
  transaction identifiers B-10
SubtransactionAwareResource interface 2-17
  commit_substransaction operatio n2-18
  commit_subtransaction operation 2-18
subtransactions 1-7, 1-12, 2-47, 2-49, 2-51, 2-57, 2-60
subtyping viii, xi

T
Terminator interface
  rollback operatio n2-8
terminator object 2-32
transacations
  resource manager 2-63
transaction abort
  see Resource interface
    rollback operatio n2-15
transaction context 2-3

Index

management of 2-6
propagation of 2-6
transaction originator1-13, 2-3, 2-7, 2-38
glossary definitio n5
transaction service
application use of 2-26
transactional client 1-4, 2-29
glossary definitio n5
transactional object 1-4
example 2-34
transactional server
defined 1-6
TransactionFactory interface 2-32
transactions
checked 2-28–2-29, 2-31
consistency propert y2-49
consistency property, glossary definitio n2
coordinator object 2-15, 2-17, 2-32, 2-33, 2-44, 2-55
distributed 2-31
durability 2-48
durability, glossary definition 2
flat 1-6, 1-7, 1-9, 2-31
flat,glossary definition 2
implicit propagation 2-32
interposition 2-40, 2-55, 2-57
interposition, glossary defintion 3
isolation 1-7, 1-9, 1-13, 2-8
isolation, glossary definition 3
propagation 2-29, 2-32, 2-35, 2-54, 2-58, 2-59, 4
propagation to resource manager B-5
recoverable object 1-5, 2-17
recoverable server 1-6, 2-30
recoverable server, glossary defintion 4
recoverable server,example 2-33
resource manager 1-9, B-6
terminator object 2-32
two-phase commit protoco l1-12, 2-14, 2-40, 2-43, 2-49, 2-56, 2-63, B-8, B-10
two-phase commit, glossary definiti on5

X
X/Open vi
X/Open TX interface B-1–B-3
X/Open XA interface B-2, 2-63

Transaction Service, v1.3
Reference Sheet

This is a revision of the Transaction Service. You will find specific changes marked with change bars and colored text.

OMG documents used to create this revision:

- Transaction Service, v1.2.1: formal/01-11-03
- RTF Report: ptc/2002-01-24
- Convenience document: ptc/2002-01-25
- OTS RTF XA Proposal: ptc/02-02-04