Transaction Service Specification

Version 1.1 -
New edition - May 2000

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 at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 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.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

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 Med*, 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*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
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.

Acknowledgments

The following companies submitted parts of the *Transaction* 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.

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 chapter also contains an appendix that explains the relationship between the Transaction Service and TP standards, and an appendix 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 which 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 2-18. 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 2.2.1.1, "Direct Context Management: Explicit Propagation," on page 2-19).

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

The following figure shows a simple application which includes these basic elements.

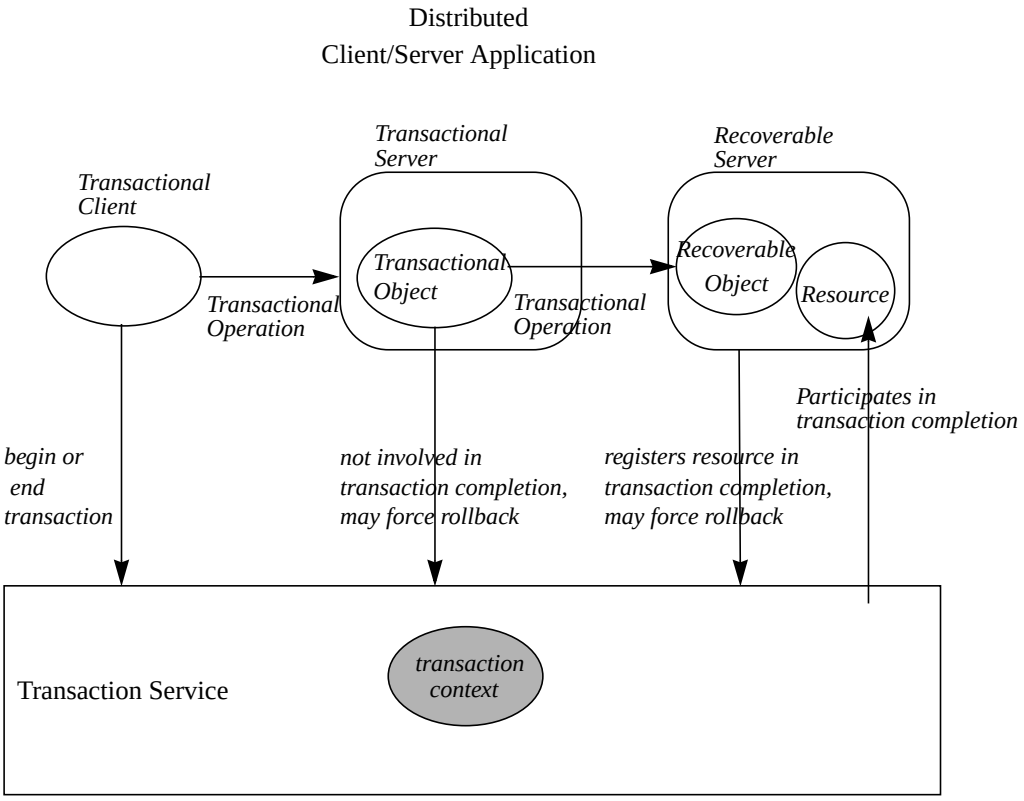

Figure 1-1 Application Including Basic Elements

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 which 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 which 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 modelled on the X/Open DTP transaction model definition.[1]

A flat transaction is considered to be a top-level transaction—see the next section—that cannot have a child transaction.

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.

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 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 which 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 which 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 by the 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.
- The ability to execute 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 which 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 which 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 which 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.

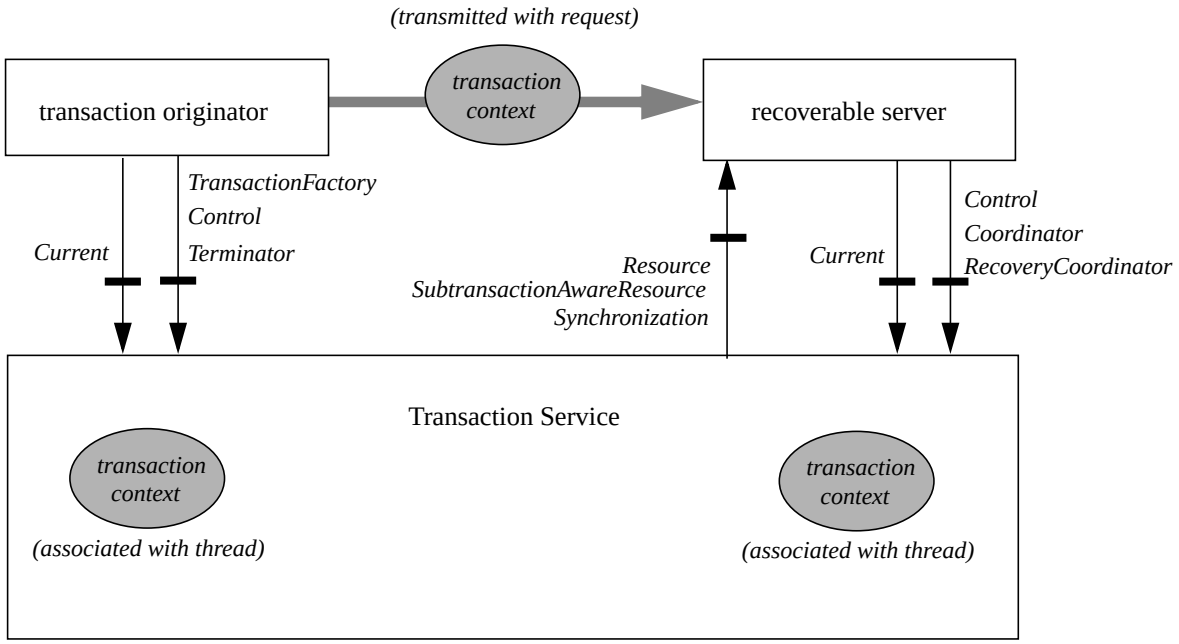

Figure 1-2 Major Components and Interfaces of the Transaction Service

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.

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 is not required to initialize the transaction context of every request handler. It is required to initialize the transaction context only if the interface supported by the target object is derived from the *TransactionalObject* interface. Otherwise, the initial transaction context of the thread is undefined.

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 Section 1.3.6.3, “WRONG_TRANSACTION Exception,” on page 1-17 for a more precise definition.)

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 implementation of the Transaction Service may restrict the ability for some or all of these objects to be transmitted to or used in other execution environments, to enable it to guarantee transaction integrity.

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
};
```

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 **CosTransactions** module adds new standard exceptions to CORBA for TRANSACTION_REQUIRED, TRANSACTION_ROLLEDBACK, and INVALID_TRANSACTION. These exceptions are defined in Chapter 3, Section 3.15 of the *Common Object Request Broker: Architecture and Specification.*

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. 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.)

1.3.6.3 WRONG_TRANSACTION Exception

The **CosTransactions** module adds the WRONG_TRANSACTION exception that can be raised by the ORB when returning the response to a deferred synchronous request. This exception is defined in Chapter 4 of the *Common Object Request Broker: Architecture and Specification.*

1.3.6.4 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 below along with the operations that raise them.

Transaction Service Interfaces 2

Contents

This chapter contains the following sections.

2.1 Introduction

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

- *Current*
- *TransactionFactory*
- *Terminator*
- *Coordinator*
- *RecoveryCoordinator*
- *Resource*
- *Synchronization*
- *Subtransaction Aware Resource*
- *Transactional Object*

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.

2.1.1 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 designed to be supported by a pseudo object whose behavior depends upon and may alter the transaction context associated with the invoking thread. It may be shared with other object services (e.g., security) and is obtained by using a resolve initial references("TransactionCurrent") operation on the `CORBA::ORB` interface. **Current** supports the following operations:

```
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);

    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.

2.1.1.1 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.

2.1.1.2 *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 2.1.4, “Terminator Interface) and Section 1.3.6, “Exceptions 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.

2.1.1.3 *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 “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.

2.1.1.4 *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 2.1.5, “Coordinator Interface,” on page 2-8).

2.1.1.5 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 2.1.5, “Coordinator Interface,” on page 2-8).

2.1.1.6 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 2.1.5, “Coordinator Interface,” on page 2-8).

2.1.1.7 set_timeout

This operation modifies a state variable associated with the target object that affects the time-out period associated with top-level transactions created by subsequent invocations of the **begin** operation. If the parameter has a nonzero 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. If the parameter is zero, then no application specified time-out is established.

2.1.1.8 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 reestablish 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.

2.1.1.9 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 reestablish 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.

2.1.1.10 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 2.1.3, "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.

2.1.2 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. A **TransactionFactory** is located using the **FactoryFinder** interface of the life cycle service and not by the **resolve_initial_reference** operation on the *ORB* interface defined in "Example Object Adapters" in Chapter 2 of the *Common Object Request Broker: Architecture and Specification.*

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

2.1.2.1 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.

2.1.2.2 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 2.3.2, "ORB/TS Implementation Considerations," on page 2-44) 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 which originated outside of the Transaction Service.

2.1.3 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 allows 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 2.1.1, “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.

2.1.3.1 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.

2.1.3.2 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.

2.1.4 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.

2.1.4.1 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.

If the **report_heuristics** parameter is true, the Transaction Service will report inconsistent or possibly inconsistent outcomes using the HeuristicMixed and HeuristicHazard exceptions (defined in Section 1.3.6, “Exceptions). A Transaction Service implementation may optionally use the Event Service to report heuristic decisions.

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.

2.1.4.2 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.

2.1.5 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.

2.1.5.1 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.

2.1.5.2 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.

2.1.5.3 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.

2.1.5.4 is_same_transaction

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

2.1.5.5 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.

2.1.5.6 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).

2.1.5.7 *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.

2.1.5.8 *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.

2.1.5.9 *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.

2.1.5.10 *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.

2.1.5.11 *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 completed. Otherwise, the resource is registered as a participant in the current transaction. If the current transaction is a subtransaction, the resource will not receive prepare or commit requests until the top-level ancestor terminates.

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

2.1.5.12 *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.

2.1.5.13 *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) has already been terminated. The standard exception TRANSACTION_ROLLEDBACK may be raised if the subtransaction (or any ancestor) has been marked rollback only.

2.1.5.14 *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.

2.1.5.15 *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.

2.1.5.16 *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 has already been prepared. 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.

2.1.5.17 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**.

2.1.6 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);
};
```

2.1.6.1 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.

2.1.7 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();
};
```

2.1.7.1 *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). 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.

2.1.7.2 *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) 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 **rollback** is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

2.1.7.3 *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) 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.

2.1.7.4 *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.

2.1.7.5 *forget*

This operation is performed only if the resource raised a heuristic outcome exception to **rollback**, **commit**, or **commit_one_phase**. 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.

2.1.8 *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. 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.

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

2.1.8.1 *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.

2.1.8.2 *after_completion*

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 commitment process.

2.1.9 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();
};
```

2.1.9.1 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.

2.1.9.2 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.

2.1.10 TransactionalObject Interface

The **TransactionalObject** interface is used by an object to indicate that it is transactional. By supporting the **TransactionalObject** interface, an object indicates that it wants the transaction context associated with the client thread to be associated with all operations on its interface.

```
interface TransactionalObject {
};
```

The **TransactionalObject** interface defines no operations. It is simply a marker.

2.2 The User's View

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

2.2.1 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.

2.2.1.1 Direct Context Management: Explicit Propagation

The client application directly accesses the **Control** object, and the other objects which 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.

2.2.1.2 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.

2.2.1.3 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.

2.2.1.4 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.

2.2.2 Interfaces

Table 2-1 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	`TransactionalObject` interface

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.

2.2.3 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.

2.2.4 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.

2.2.4.1 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.

2.2.4.2 Commit Check

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

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

2.2.4.3 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.

2.2.5 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.

2.2.6 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.

2.2.6.1 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 which have the same hash code as the coordinator of the current request.

2.2.6.2 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 2.3.1, "Transaction Service Protocols," on page 2-33.

2.2.6.3 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 which apply to the transaction as a whole, rather than to individual requests to the server.

2.2.7 Application Portability

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

2.2.7.1 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.

2.2.7.2 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 which support unchecked behavior.

2.2.8 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.

2.2.9 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.

2.2.10 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.

2.2.10.1 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 **commits** 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);
...
```

2.2.10.2 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);
```

2.2.10.3 *Example of a Recoverable Server*

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

```
interface BankAccount1:
      CosTransactions::TransactionalObject,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
};
```

2.2.10.4 *Example of a Transactional Object*

BankAccount2 is an object with external resources that inherits from the **TransactionalObject** interface:

```
interface BankAccount2: CosTransactions::TransactionalObject
{
...
  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
```

2.2.11 *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.

2.2.11.1 *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.

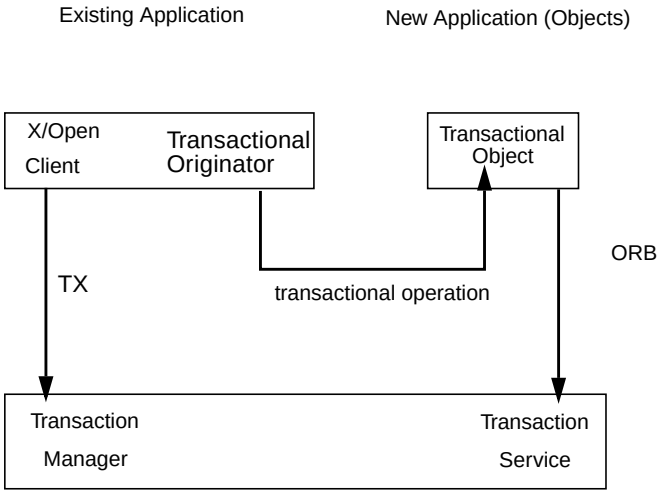

Figure 2-3 X/Open Client

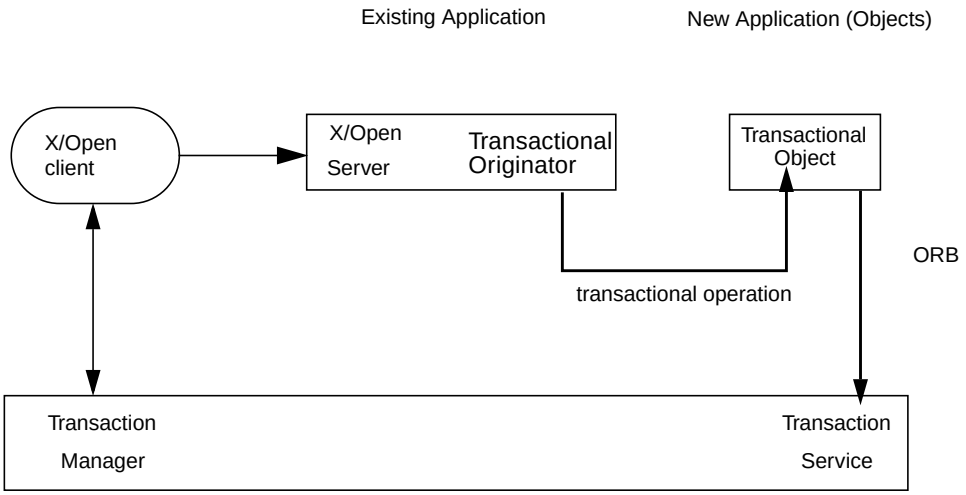

Figure 2-4 X/Open Server

2.2.11.2 *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.

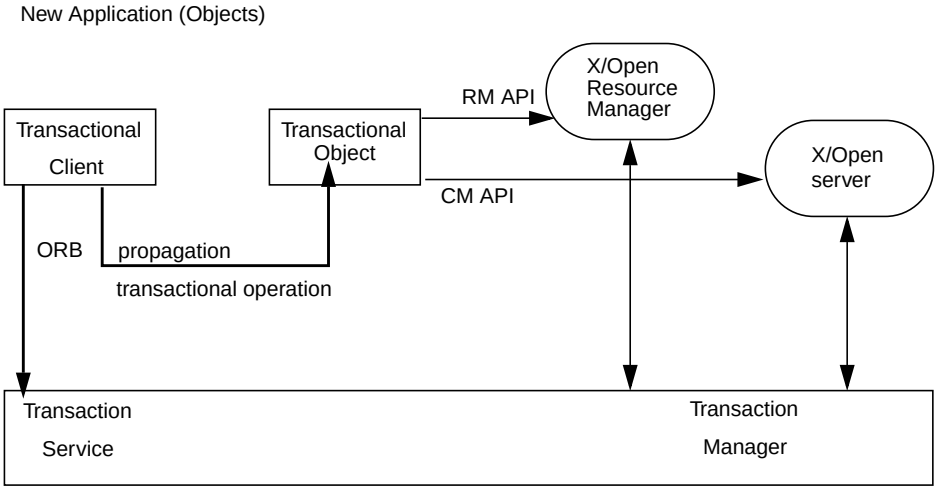

Figure 2-5 Sample Transaction Managed by the Transaction Service

2.2.11.3 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.

2.2.12 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 2.3.1, “Transaction Service Protocols,” on page 2-33.

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.

2.2.12.1 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).

2.2.12.2 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**.

2.2.12.3 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 2.3.1, "Transaction Service Protocols," on page 2-33.