

X/Open Preliminary Specification

Common Object Services, Volume 2

X/Open Company Ltd.



© October 1995, X/Open Company Limited

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

This work is published by X/Open Company Ltd., U.K. under the terms of its agreement with the Object Management Group. Ownership of the intellectual property rights remain vested with the Object Management Group and the authors listed here:

AT&T/NCR Corporation	Iona Technologies, Ltd.	Servio Corporation
BNR Europe Limited	Itasca Systems, Inc.	Siemens Nixdorf Informationssysteme AG
Digital Equipment Corporation	Novell, Inc.	SunSoft, Inc.
Groupe Bull	O2 Technology, SA	Tandem Computers, Inc.
Hewlett-Packard Company	Object Design, Inc.	Teknekron Software Systems, Inc.
HyperDesk Corporation	Objectivity, Inc.	Tivoli Systems, Inc.
ICL plc	Ontos, Inc.	Transarc Corporation
Ing. C. Olivetti & C. SpA	Oracle Corporation	Versant Object Technology Corporation
IBM Corporation	Persistence Software, Inc.	

This document is equivalent to parts of OMG Document Number 94-1-1.

X/Open Preliminary Specification

Common Object Services, Volume 2

ISBN: 1-85912-125-X

X/Open Document Number: P502

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

Chapter 1	Introduction.....	1
1.1	Persistent Object Service	1
1.2	Transaction Service.....	1
1.3	Concurrency Control Service	2
1.4	Relationship Service	2
1.5	Externalization Service	3
Chapter 2	General Design Principles	5
2.1	Service Design Principles	5
2.1.1	CORBA Concepts	5
2.1.2	Basic Flexible Services.....	5
2.1.3	Generic Services	5
2.1.4	Local and Remote Implementations.....	6
2.1.5	Quality of Service.....	6
2.1.6	Objects Conspire	6
2.1.7	Use of Callback Interfaces.....	7
2.1.8	Global Identifier Spaces.....	7
2.1.9	Finding and Using Services	8
2.2	Interface Style Consistency	9
2.2.1	Exceptions and Return Codes	9
2.2.2	Explicit versus Implicit Operations	9
2.2.3	Interface Inheritance.....	9
2.3	Key Design Decisions.....	10
2.3.1	Naming Service Issues.....	10
2.3.2	Universal Object Identity	10
2.4	Integration with Future Object Services	11
2.4.1	Archive Service.....	11
2.4.2	Backup/Restore Service	11
2.4.3	Change Management Service.....	12
2.4.4	Data Interchange Service.....	12
2.4.5	Implementation Repository Service	12
2.4.6	Interface Repository Service	12
2.4.7	Internationalization Service.....	12
2.4.8	Logging Service	12
2.4.9	Query Service.....	13
2.4.10	Recovery Service	14
2.4.11	Replication Service	14
2.4.12	Security Service	14
2.4.13	Startup Service.....	14
2.5	Service Dependencies	15
2.5.1	Naming Service	15
2.5.2	Event Service.....	15

2.5.3	Life Cycle Service.....	15
2.5.4	Persistent Object Service	15
2.5.5	Relationship Service	15
2.5.6	Externalization Service	16
2.5.7	Transaction Service.....	16
2.5.8	Concurrency Control Service	17
2.6	Relationship to CORBA.....	18
2.6.1	ORB Interoperability Considerations	18
2.7	Relationship to Object Model.....	20
2.8	Conformance to Existing Standards	20
Chapter 3	Persistent Object Service Specification.....	21
3.1	Introduction	21
3.2	Goals and Properties	23
3.2.1	Basic Capabilities	23
3.2.2	Object-oriented Storage	23
3.2.3	Open Architecture	24
3.2.4	Views of Service	25
3.3	Service Structure	27
3.4	The CosPersistencePID Module	29
3.4.1	The PID Interface	29
3.4.2	Example PIDFactory Interface	30
3.5	The CosPersistencePO Module.....	32
3.5.1	The PO Interface.....	32
3.5.2	The POFactory Interface.....	34
3.5.3	The SD Interface	34
3.6	The CosPersistencePOM Module.....	35
3.7	Persistent Data Service (PDS) Overview	38
3.8	The CosPersistencePDS Module.....	39
3.9	The Direct Access (PDS_DA) Protocol	40
3.10	The CosPersistencePDS_DA Module	41
3.10.1	The PID_DA Interface.....	42
3.10.2	The Generic DAObject Interface.....	42
3.10.3	The DAObjectFactory Interface	43
3.10.4	The DAObjectFactoryFinder Interface	43
3.10.5	The PDS_DA Interface	43
3.10.6	Defining and Using DA Data Objects.....	44
3.10.7	The DynamicAttributeAccess Interface	46
3.10.8	The PDS_ClusteredDA Interface	46
3.11	The ODMG-93 Protocol	48
3.12	The Dynamic Data Object (DDO) Protocol	49
3.13	The CosPersistenceDDO Module.....	51
3.14	Other Protocols.....	53
3.15	Datastores: The CosPersistenceDS_CLI Module	54
3.15.1	The UserEnvironment Interface	56
3.15.2	The Connection Interface	56
3.15.3	The ConnectionFactory Interface	56
3.15.4	The Cursor Interface.....	57

3.15.5	The CursorFactory Interface	57
3.15.6	The PID_CLI Interface	57
3.15.7	The Datastore_CLI Interface	58
3.16	Other Datastores	61
3.17	Standards Conformance	61
Chapter 4	Concurrency Control Service Specification	63
4.1	Service Description.....	63
4.1.1	Basic Concepts of Concurrency Control.....	63
4.2	Locking Model.....	66
4.2.1	Lock Modes	66
4.2.2	Multiple Possession Semantics	67
4.3	Two-phase Transactional Locking.....	68
4.4	Nested Transactions	69
4.5	The CosConcurrencyControl Module.....	70
4.5.1	Types and Exceptions	71
4.5.2	The LockCoordinator Interface.....	72
4.5.3	The LockSet Interface.....	72
4.5.4	The TransactionalLockSet Interface	73
4.5.5	The LockSetFactory Interface	74
Chapter 5	Externalization Service Specification	75
5.1	Service Description.....	75
5.2	Service Structure	76
5.2.1	Client Model of Externalization.....	76
5.2.2	Stream Model of Externalization.....	76
5.2.3	Object Model of Externalization.....	77
5.2.4	Object Model of Internalization.....	79
5.3	Object and Interface Hierarchies	82
5.4	Interface Summary	84
5.5	The CosExternalization Module.....	86
5.5.1	The StreamFactory Interface.....	86
5.5.2	The FileStreamFactory Interface.....	87
5.5.3	The Stream Interface	87
5.6	The CosStream Module	89
5.6.1	The StreamIO Interface.....	90
5.6.2	The Streamable Interface.....	91
5.6.3	The StreamableFactory Interface	92
5.7	The CosCompoundExternalization Module.....	93
5.7.1	The Node Interface	94
5.7.2	The Role Interface	95
5.7.3	The Relationship Interface	95
5.7.4	The PropagationCriteriaFactory Interface	96
5.8	Specific Externalization Relationships	97
5.9	The CosExternalizationContainment Module.....	98
5.10	The CosExternalizationReference Module.....	99
5.11	Standard Stream Data Format.....	100
5.11.1	Externalized Repeated Reference Data	101

	5.11.2	Externalized NIL Data	101
Chapter 6		Relationship Service Specification	103
	6.1	Service Description.....	103
	6.1.1	Key Features of the Relationship Service.....	104
	6.1.2	The Relationship Service versus CORBA Object References	105
	6.1.3	Resolution of Technical Issues	106
	6.2	Service Structure	108
	6.2.1	Levels of Service.....	108
	6.2.2	Hierarchy of Relationship Interface.....	111
	6.2.3	Hierarchy of Role Interface.....	111
	6.2.4	Interface Summary	112
	6.3	The Base Relationship Model.....	114
	6.3.1	Relationship Attributes and Operations.....	115
	6.3.2	Higher Degree Relationships	115
	6.3.3	Operations.....	117
	6.3.4	Consistency Constraints.....	119
	6.3.5	Implementation Strategies.....	119
	6.3.6	The CosObjectIdentity Module.....	119
	6.3.6.1	The IdentifiableObject Interface	120
	6.3.7	The CosRelationships Module.....	120
	6.3.7.1	The RelationshipFactory Interface	123
	6.3.7.2	The Relationship Interface	124
	6.3.7.3	The Role Interface	125
	6.3.7.4	The RoleFactory Interface	128
	6.3.7.5	The RelationshipIterator Interface	129
	6.4	Graphs of Related Objects.....	130
	6.4.1	Graph Architecture.....	130
	6.4.2	Traversing Graphs of Related Objects.....	132
	6.4.3	Compound Operations.....	133
	6.4.4	An Example Traversal Criteria.....	134
	6.4.5	The CosGraphs Module	135
	6.4.5.1	The TraversalFactory Interface	137
	6.4.5.2	The Traversal Interface	138
	6.4.5.3	The TraversalCriteria Interface	138
	6.4.5.4	The Node Interface	139
	6.4.5.5	The NodeFactory Interface	140
	6.4.5.6	The Role Interface	140
	6.4.5.7	The EdgeIterator Interface	141
	6.5	Specific Relationships	142
	6.5.1	Containment and Reference	142
	6.5.2	The CosContainment Module.....	142
	6.5.3	The CosReference Module.....	143
Chapter 7		Transaction Service Specification	145
	7.1	Service Description.....	145
	7.1.1	Overview of Transactions	145
	7.1.2	Transactional Applications	146

7.1.3	Definitions	147
7.1.4	Transaction Service Functionality	149
7.1.5	Principles of Function, Design and Performance	151
7.2	Service Architecture	155
7.2.1	Typical Usage.....	156
7.2.2	Transaction Context	156
7.2.3	Context Management.....	157
7.2.4	Data Types.....	157
7.2.5	Exceptions	158
7.2.5.1	Standard Exceptions.....	158
7.2.5.2	Heuristic Exceptions	158
7.2.5.3	Other Exceptions.....	159
7.3	Transaction Service Interfaces.....	160
7.3.1	The Current Interface	160
7.3.2	The TransactionFactory Interface	163
7.3.3	The Control Interface	163
7.3.4	The Terminator Interface.....	164
7.3.5	The Coordinator Interface.....	165
7.3.6	The RecoveryCoordinator Interface	168
7.3.7	The Resource Interface	168
7.3.8	The SubtransactionAwareResource Interface	170
7.3.9	The TransactionalObject Interface.....	171
7.4	The User View	172
7.4.1	Application Programming Models	172
7.4.1.1	Direct Context Management	172
7.4.1.2	Indirect Context Management	173
7.4.2	Interfaces.....	174
7.4.3	Checked Transaction Behaviour	175
7.4.4	X/Open Checked Transactions.....	175
7.4.5	Implementing a Transactional Client: Heuristic Completions	176
7.4.6	Implementing a Recoverable Server.....	176
7.4.7	Application Portability	177
7.4.8	Distributed Transactions	177
7.4.9	Applications Using Both Checked and Unchecked Services.....	178
7.4.10	Examples.....	178
7.4.11	Model Interoperability.....	180
7.4.12	Failure Models.....	183
7.4.12.1	Transaction Originator.....	183
7.4.12.2	Transactional Server	183
7.4.12.3	Recoverable Server	184
7.5	The Implementor View.....	185
7.5.1	Transaction Service Protocols.....	185
7.5.1.1	General Principles.....	185
7.5.1.2	Normal Transaction Completion.....	186
7.5.1.3	Failure and Recovery	191
7.5.1.4	Transaction Completion after Failure.....	192
7.5.2	ORB/TS Implementation Considerations	194
7.5.2.1	Transaction Propagation	194

7.5.2.2	Transaction Service Interoperation	195
7.5.2.3	Transaction Service Portability	197
7.5.2.4	The Transaction Service Callbacks	199
7.5.2.5	Behaviour of the Callback Interfaces	200
7.5.3	Model Interoperability	200
7.6	The CosTransactions Module	202
7.6.1	The CosTSInteroperation Module	204
7.6.2	The CosTSPortability Module	205
Appendix A	The Transaction Service and TP Standards	207
A.1	Support of X/Open TX interface	207
A.1.1	Requirements	207
A.1.2	TX Mappings	207
A.2	Support of X/Open Resource Managers	209
A.2.1	Requirements	209
A.2.2	XA Mappings	209
A.2.3	XID	209
A.2.3.1	Interactions with an XA-compliant RM	210
A.3	Interoperation with Transactional Protocols	213
A.3.1	OSITP Interoperability	213
A.3.2	SNA LU 6.2 Interoperability	214
A.3.3	ODMG Standard	216
A.4	ODMG Model	217
	Glossary	219
	Index	225

List of Figures

2-1	Event Channel Managing Multiple Simultaneous Consumer Clients	7
3-1	Roles in the Persistent Object Service	21
3-2	Major Components of the POS and their Interactions	28
3-3	Example to Illustrate POM Functions	37
3-4	Direct Access Protocol Interfaces	40
3-5	Structure of a DDO	50
5-1	Externalization Control Flow (Streamable Object is Not a Node)	78
5-2	Externalization Control Flow (Streamable Object is a Node)	79
5-3	Internalization Control Flow (Streamable Object is Not a Node)	80
5-4	Internalization Control Flow (Streamable Object is a Node)	81
5-5	Object Externalization Service Booch Class (=Interface)	83
5-6	Internalizing a Node Returns New Object and Corresponding Roles	94
6-1	Base Relationships	108
6-2	Navigation Functionality of Base Relationships	109
6-3	Example Graph of Related Objects	110
6-4	Relationship Interface Hierarchy	111

6-5	Role Interface Hierarchy	111
6-6	Simple Relationship Type: Documents Reference Books.....	114
6-7	Simple Relationship Instance: My Document References the Book War and Peace	115
6-8	Satisfactory Ternary Check-out Relationship	116
6-9	Unsatisfactory Ternary Check-out Relationship.....	116
6-10	Another Unsatisfactory Representation	117
6-11	Creating a Role for an Object	117
6-12	Fully Established Binary Relationship.....	118
6-13	Two Binary One-to-many Containment Relationships	123
6-14	Example Graph of Related Objects	131
6-15	Traversal of a Graph for a Compound copy() Operation.....	134
6-16	deep, shallow and none Propagation Values	135
7-1	Application Including Definitions	147
7-2	Major Components and Interfaces of the Transaction Service.....	155
7-3	X/Open Client.....	181
7-4	X/Open Server	181
7-5	Example Transaction Export	182
7-6	Model Interoperability Example.....	201

List of Tables

4-1	Lock Compatibility.....	67
5-1	Client Functional Interfaces.....	84
5-2	Service Construction Interfaces	84
5-3	Compound Externalization Interfaces	85
6-1	Interfaces Defined in the CosObjectIdentity Module.....	112
6-2	Interfaces Defined in the CosRelationship Module.....	112
6-3	Interfaces Defined in the CosGraph Module	113
6-4	Interfaces Defined in the CosReferences Module	113
6-5	Interfaces Defined in the CosContainments Module.....	113
7-1	Use of Transaction Service Functionality.....	174

Preface

X/Open

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

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to `info-server@xopen.co.uk` with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document defines the common object services for persistent objects, transactions, concurrency control, relationships and externalisation. It is equivalent to parts of OMG Document Number 94-1-1.

Structure

This document is organised as follows:

- Chapter 1 provides a summary of key features of each service.
- Chapter 2 describes the design principles used in this specification. It addresses:
 - service dependencies
 - relationship to CORBA
 - relationship to the OMG Object Model
 - standards conformance.
- Chapter 3 describes the Persistent Object Service specification.
- Chapter 4 describes the Transaction Service specification.
- Chapter 5 describes the Concurrency Control Service specification.
- Chapter 6 describes the Relationship Service.
- Chapter 7 describes the Externalization Service.

Each service definition chapter begins with an overview, and includes sections on design principles and resolution of technical issues (raised in the OMG Object Services Architecture).

Appendix A describes the relationship of the transaction service to TP standards.

A glossary and index are included at the end.

Typographical Conventions

The following typographical conventions are used throughout this document:

Helvetica Pseudo-IDL language elements.

Helvetica bold IDL language and syntax elements.

Courier C-language elements.

Code examples written in pseudo-IDL and C are further identified by means of a comment; unidentified examples are written in IDL.

Trade Marks

COSS™ is a trade mark of the Object Management Group, Inc.

OMG® and Object Management® are registered trade marks of the Object Management Group, Inc.

X/Open® is a registered trade mark, and the “X” device is a trade mark, of X/Open Company Limited.

Referenced Documents

The following X/Open documents are referenced in this specification:

CLI

X/Open CAE Specification, March 1995, Data Management: SQL Call Level Interface (CLI) (ISBN: 1-85912-081-4, C451).

CORBA 1.2

X/Open CAE Specification, July 1994, The Common Object Request Broker: Architecture and Specification (ISBN: 1-85912-044-X, C432), in conjunction with the Object Management Group (OMG).

CORBA 2.0

X/Open Preliminary Specification, October 1995, The Common Object Request Broker: Architecture and Specification (ISBN: 1-85912-140-3, P431), in conjunction with the Object Management Group (OMG).

COS, Volume 1

X/Open Preliminary Specification, July 1994, Common Object Services, Volume 1 (ISBN: 1-85912-482-2, P432), in conjunction with the Object Management Group (OMG).

CPI-C, Version 2

X/Open CAE Specification, October 1995, Distributed Transaction Processing: The CPI-C Specification, Version 2 (ISBN: 1-85912-135-7, C419).

DTP

X/Open Guide, November 1993, Distributed Transaction Processing: Reference Model, Version 2 (ISBN: 1-85912-019-9, G307).

TX

X/Open CAE Specification, April 1995, Distributed Transaction Processing: The TX (Transaction Demarcation) Specification (ISBN: 1-85912-094-6, C504).

TxRPC

X/Open CAE Specification, October 1995, Distributed Transaction Processing: The TxRPC Specification (ISBN: 1-85912-115-2, C505).

XA

X/Open CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification (ISBN: 1-872630-24-3, C193 or XO/CAE/91/300).

XATMI

X/Open CAE Specification, October 1995, Distributed Transaction Processing: The XATMI Specification (ISBN: 1-85912-130-6, C506).

The following non-X/Open documents are referenced in this specification:

- IDAPI Working Draft, Borland International, August 1993.
- ISO/IEC 10026-3:1992 Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 3: Protocol Specification.
- ITU-T X.900 Series, ISO/IEC 10746, Reference Model — Open Distributed Processing
- ITU-T X.Trader, ISO/IEC 13235, Open Distributed Processing Trader.

Referenced Documents

- J.E.B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*.
- J.N. Gray, *Notes on Database Operating Systems* in *Operating Systems: An Advanced Course*, Ed. Bayer, Graham and Seegmuller.
- J.N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*.
- James Rumbaugh, *Controlling Propagation of Operations using Attributes on Relations*, OOPSLA 1988 Proceedings.
- James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, *Object-oriented Modelling and Design*, Prentice Hall 1991.
- *Object Services Architecture*, Document Number 92-8-4, Object Management Group, Framingham, MA 1992.
- Microsoft Open Database Connectivity Software Development Kit, Programmer Reference, Version 1.0, Microsoft Corporation, 1992.
- P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*.
- Richard M. Soley, Ph.D., *Object Management Architecture Guide*, Revision 2.0, Second Edition, Ed., Object Management Group, Inc., Framingham, MA 1992.
- R.G.G. Cattell, T. Atwood, J. Duhl, G. Ferran, M. Loomis, D. Wade, *The Object Database Standard: ODMG-93*, Object Database Management Group, Morgan Kaufmann, 1993.
- Systems Network Architecture: LU 6.2 Reference: Peer Protocols, Order Number SC31-6806, International Business Machines Corporation.

This chapter provides a summary of the key features described in this specification.

1.1 Persistent Object Service

The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects.

The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. A major feature of the Persistent Object Service is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where mechanisms useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers do not apply to mainframes.

1.2 Transaction Service

The Transaction Service supports multiple transaction models, including the flat (mandatory) and nested (optional) models.

The Transaction Service supports interoperability between different programming models. For example, some users want 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 the object and procedural code to share a single transaction.

Network interoperability is also supported, since users need communication between different systems, including the ability to have one transaction service interoperate with a cooperating transaction service using different ORBs.

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

The Transaction Service can be implemented in a TP monitor environment, so it supports the ability to execute multiple transactions concurrently, and to execute clients, servers and transaction services in separate processes.

1.3 Concurrency Control Service

The Concurrency Control Service enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

Concurrent use of a resource is regulated with locks. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the client's activities might conflict. Hence, a client must obtain an appropriate lock before accessing a shared resource. The Concurrency Control Service defines several lock modes, which correspond to different categories of access. This variety of lock modes provides flexible conflict resolution. For example, providing different modes for reading and writing lets a resource support multiple concurrent clients on a read-only transaction. The Concurrency Control Service also defines intention locks that support locking at multiple levels of granularity.

1.4 Relationship Service

The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The Relationship Service defines two new kinds of objects: *relationships* and *roles*. A role represents a CORBA object in a relationship. The relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the Role interface can be extended to add role-specific attributes and operations.

Type and cardinality constraints can be expressed and checked. Exceptions are raised when the constraints are violated.

The Life Cycle Service defines operations to copy, move and remove graphs of related objects, while the Relationship Service allows graphs of related objects to be traversed without activating the related objects.

Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references: role objects can be located with their objects and need not depend on a centralised repository of relationship information. As such, navigating a relationship can be a local operation.

The Relationship Service supports the compound life cycle component of the Life Cycle Service by defining object graphs.

1.5 Externalization Service

The Externalization Service defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so on) and then be internalized into a new object in the same or a different process. The externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. For portability, clients can request that externalized data be stored in a file whose format is defined with the Externalization Service specification.

The Externalization Service is related to the Relationship Service and parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for facilities, directory services and file services.

General Design Principles

This chapter describes the design principles used in this specification.

2.1 Service Design Principles

2.1.1 CORBA Concepts

The service designs use and build 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 specialise 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 can and 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 and entities.

- Do not build non-type properties into interfaces.

2.1.2 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 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, Event and Relationship Services as building blocks.

2.1.3 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.

2.1.4 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 implementation. This allows considerable flexibility with regard to 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.

2.1.5 Quality of Service

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.

2.1.6 Objects Conspire

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** or **EventChannel** interface. Hidden to all 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. This is shown in Figure 2-1 on page 7.

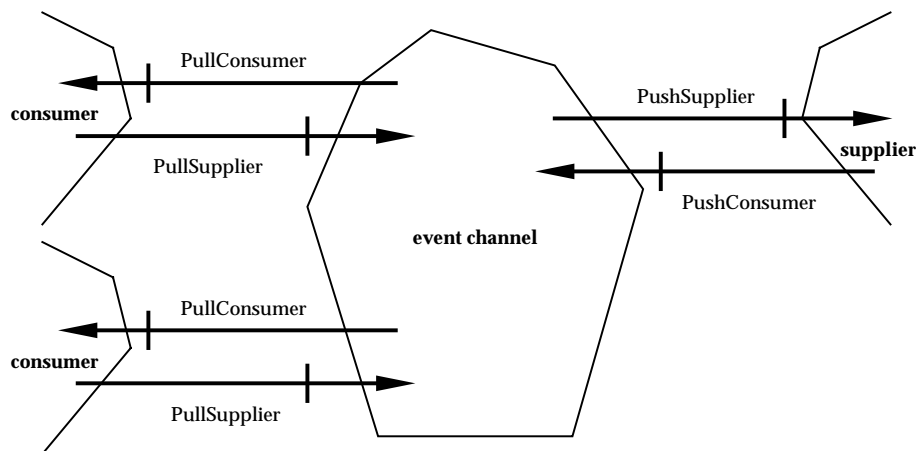


Figure 2-1 Event Channel Managing Multiple Simultaneous Consumer Clients

The graphical notation shown in Figure 2-1 is used throughout this document and in other service specifications. An arrow with a vertical bar is used to show that the target object supports the interface named adjacent to it, and that clients holding an object reference to it of this type can invoke operations. In shorthand, the object reference (held by the client) supports the interface. The arrow points from the client to the target (server) object. A closed irregular shape indicates a conspiracy of one or more objects. In other words, it corresponds to a conceptual object that may be composed of one or more CORBA objects that together provide some coordinated service to potentially multiple clients making requests using different object references.

2.1.7 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.

2.1.8 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.

2.1.9 Finding and Using Services

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 by means of 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.

2.2 Interface Style Consistency

2.2.1 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 by means of output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

2.2.2 Explicit versus Implicit Operations

Operations are always explicit rather than implied; for example, 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.

2.2.3 Interface Inheritance

Interface inheritance (subtyping) is used whenever it appears 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.

2.3 Key Design Decisions

2.3.1 Naming Service Issues

Distinct from Property and Trading Services, naming contexts have some similarity to property lists; that is, lists of values associated with objects though not necessarily part of the object's state, and the Naming Service in general, have elements in common with a Trading Service. However, following the *bauhaus* principle of keeping services as simple and as orthogonal as possible, these services have been kept distinct and are being addressed separately.

2.3.2 Universal Object Identity

These services do not require the concept of universal object identity.

2.4 Integration with Future Object Services

This section discusses how the object services could evolve to integrate with future services, such as:

- Archive
- Backup/Restore
- Change Management (Versioning)
- Data Interchange
- Implementation Repository
- Interface Repository
- Internationalization
- Logging
- Query
- Recovery
- Replication
- Security
- Startup.

2.4.1 Archive Service

Persistent Object Service. The Archive Service copies objects from an active/persistent store to a backup store and *vice versa*. This service should be able to archive objects stored with the Persistent Object Service.

Externalization Service. The Archive Service copies objects from an active/persistent store to a backup store and *vice versa*. This service could use the Externalization Service to get the internal state of objects for saving and to subsequently recreate objects with this stored state. If only persistent objects need to be archived, then the Persistent Object Service could be used instead.

2.4.2 Backup/Restore Service

Externalization Service. The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Externalization Service as an underlying mechanism for objects, regardless of whether they are persistent.

Persistent Object Service. The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Persistent Object Service as an underlying mechanism for persistent objects.

Transaction Service. The permanence of effect property of a transaction implies that the state established by the commitment of a transaction will not be lost. To guarantee this property, the storage media on which the objects updated by the transaction are stored must be backed-up to secondary storage to ensure that they are not lost should the primary storage media fail. Similarly, the storage media used by the logging service must be restorable should the media fail. Since there are multiple components which require backup services, a single interface would be advantageous.

2.4.3 Change Management Service

Persistent Object Service. The Change Management Service supports the identification and consistent evolution of objects including version and configuration management. This service should work with the Persistent Object Service to allow persistent objects to evolve from the old to new versions.

2.4.4 Data Interchange Service

Persistent Object Service. The Data Interchange Service enables objects to exchange some or all of their associated state. This service should work with the Persistent Object Service to allow state to be exchanged when one or more of the objects are persistent.

Externalization Service. The Data Interchange Service enables objects to exchange some or all of their associated state. This service could use the Externalization Service to allow state to be exchanged regardless of whether the objects are persistent.

2.4.5 Implementation Repository Service

Persistent Object Service. The Implementation Repository Service supports the management of object implementations. The Persistent Object Service may depend on this to determine what persistent data an object contains. This dependency is at the implementation level.

2.4.6 Interface Repository Service

Persistent Object Service. The Interface Repository Service supports run-time access to OMG IDL-specified definitions such as object interfaces and type definitions. The Persistent Object Service depends on this to determine whether a persistent object supports certain interfaces.

2.4.7 Internationalization Service

Naming Service. Naming Service interfaces may also need to be extended (for example, the structure of names extended, additional name resolution operations added) to better support representing and resolving names for some languages and cultures.

2.4.8 Logging Service

Transaction Service. A Logging Service implements the abstract notion of an infinitely long, sequentially-accessible, append-only file. It typically supports multiple log files, where each log file consists of a sequence of log records. New log records are written to the end of a log file, old log records can be read from any position in the file. To stop log files from growing too large for the underlying storage medium, a log service must provide an operation to archive old log records to allow the log file to be truncated.

Various components of a transaction processing system may require the services of a log service:

- **Transaction Service:** During the two-phase commit protocol the Transaction Service must log its state to ensure that the outcome of the committing transaction can be determined should there be a failure.
- **Recoverable (transactional) objects:** A log can be used to record old and new versions of a recoverable object for the purposes of supporting recovery.
- **Locking Service:** A log can be used to record the locks held on an object at prepare time to facilitate recovery.

Since there are multiple components within a distributed transaction processing system that require the services of a log service, a single log service interface (and potentially server) that is shared between the components is clearly advantageous.

The correctness of a Transaction Service depends upon the services of a Logging Service. For this reason, the Logging Service must meet the following requirements:

1. Restart

A restart facility allows rapid recovery from the cold start of an application. The Recovery Service used by the application (indirectly through the application's use of recoverable objects) would use the restart facility to establish a *checkpoint*; a consistent point in the execution state of the application from which the recovery process can proceed. In the absence of a checkpoint the Recovery Service would have to scan the entire log to ensure restart recovery occurs correctly.

2. Buffering and forcing operations

A Logging Service should provide two classes of operation for writing log records:

- An operation to buffer a log record (the record is not written directly to the underlying storage medium). Used during the execution of a transaction. Since the log record is buffered, the write is inexpensive.
- An operation to force a log record to the underlying storage medium. Used during the two-phase commit protocol to guarantee the correctness of the transaction. Forcing a log record also flushes all previously written, but buffered, log records.

3. Robustness

The log service should ensure the consistency of the underlying storage medium in which log files are stored. This usually involves the log service employing protocols that update the storage in a manner that would not result in the loss of any existing data (that is, careful updates), along with support for mirroring the storage media to tolerate media failures.

4. Archival

A Logging Service should provide support for archiving log records. Archival is necessary to allow the log to be truncated to ensure that it does not grow without bounds.

5. Efficiency

Since the Logging Service may be written to by multiple components within a transaction, the addition of log records must be efficient to avoid the bandwidth of log from becoming a bottleneck in the system.

2.4.9 Query Service

Persistent Object Service. The Query Service supports operations on sets and collections of objects which may result in sets or collections of objects. This service should work with the Persistent Object Service to support queries on persistent objects.

2.4.10 Recovery Service

Transaction Service. As recoverable objects are updated during a transaction, they (as Resource Managers) keep a record of the changes made to their state that is sufficient to undo the updates should the transaction rollback. The component responsible for this task is termed the Recovery Service. Various different forms of recovery are possible; however, the most common form is called value logging and involves the recoverable object recording both the old and new values of the object. When a transaction is recovered due to failure, the old value of an object is used to undo changes made to the object during the transaction. Most recovery services employ the services of a Logging Service to maintain the undo information.

2.4.11 Replication Service

Persistent Object Service. The Replication Service provides for the explicit replication of objects in a distributed environment and for the management of consistency of replicated copies. This service could use the Persistent Object Service to manage persistent replicas.

2.4.12 Security Service

Naming Service. Name resolution has been factored into the design in anticipation of security failures managed by a Security Service. The introduction of ACLs into the model should not effect existing clients of the Naming Service OMG IDL interfaces.

Persistent Object Service. The Security Service provides access control, encryption and audit control for objects and interfaces. This service should work with the Persistent Object Service to provide access control on persistent objects.

2.4.13 Startup Service

Persistent Object Service. The Startup Service supports bootstrapping and termination of object services. This service should support bootstrapping and termination of the Persistent Object Service.

2.5 Service Dependencies

The interface designs of all the services are general in nature and do not presume or require the existence of specific supporting software in order to implement them. An implementation of the Naming Service, for example, could use naming or directory services provided in a general-purpose networking environment. For example, an implementation may be based on the naming services provided by ONC or DCE. Such an implementation could provide enterprise-wide naming services to both object-based and non-object-based clients. Object-based software would see such services through the use of NamingContext objects.

Although the object services do not depend upon specific software, some dependencies and relationships do exist between services.

2.5.1 Naming Service

The Naming Service does not depend upon other services.

2.5.2 Event Service

The Event Service does not depend upon other services.

2.5.3 Life Cycle Service

Interfaces for the Life Cycle Service depend on the Naming Service.

The Life Cycle Service also defines compound operations that depend on the Relationship Service for the definition of object graphs. Appendix C, Life Cycle Operations on Distributed Object Graphs of Common Object Services, Volume 1 describes the topic of compound life cycle, and its dependence on the Relationship Service, in detail.

2.5.4 Persistent Object Service

The Externalization Service provides functions that provide for the transformation of an object into a form suitable for storage on an external media or for transfer between systems. The Persistent Object Service uses this service as a POS protocol.

The Life Cycle Service provides operations for managing object creation, deletion, copy and equivalence. The Persistent Object Service depends on this service for creating and deleting all required objects.

The Naming Service provides mappings between user-comprehensible names and CORBA object references. The Persistent Object Service depends on this service to obtain the object reference of, say, a PDS from its name or ID.

2.5.5 Relationship Service

The Relationship Service does not depend on other services. Note especially that the Relationship Service does not depend on any common storage service.

For guidelines about when to use the Relationship Service and when to use CORBA object references, refer to Section 6.1.2 on page 105.

2.5.6 Externalization Service

The Externalization Service works with the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations. Specifically, this service uses the Life Cycle Service to create and remove **Stream** and **StreamFactory** objects. ORB services may be used in **Stream** implementations to identify **InterfaceDef** and **ImplementationDef** objects corresponding to an externalized object, and to support finding an appropriate factory for recreating that object at internalization time.

The Externalization Service can also work with the Relationship Service. Implementations of **Stream** and **StreamIO** operations could use the Relationship Service to ensure that multiple references to the same object or circular references do not result in duplication of objects at internalization time or in the external representation.

In addition, the Externalization Service adds compound externalization semantics to the containment and reference relationships in the Relationship Service. Detailed information is provided in Section 5.8 on page 97.

2.5.7 Transaction Service

As concurrent requests are processed by an object, a mechanism is required to mediate access. This is necessary to provide the transaction property of isolation. The Concurrency Control Service is one possible implementation of a Locking Service.

The Transaction Service depends upon the Concurrency Control Service in the following ways:

- The Concurrency Control Service must support transaction duration locks, which provide isolation of concurrent requests by different transactions.
- The Concurrency Control Service must record transaction duration locks on persistent media, such as a log, as part of the prepare phase of commitment.
- If nested transactions are supported by the Transaction Service, then the Concurrency Control Service must also support locks that provide isolation between siblings in a transaction family and provide inheritance of locks owned by a subtransaction to its parent when the subtransaction commits.
- Transactional clients of the Concurrency Control Service are responsible for ensuring that all locks held by a transaction are dropped after all recovery or commitment operations have taken place. The **drop_locks()** operation is provided by the **LockCoordinator** interface for this purpose.

The Transaction Service supports atomicity and durability properties through the Persistent Object Service. The Transaction Service can work with the Persistent Object Service to support atomic execution of operations on persistent objects. Transactions and persistence are not provided by the same service. When coordination of multiple state changes is required to persistent data, a Persistent Object Service requires a Transaction Service. The Persistent Object Service can be used to provide persistence, but its implementation will need to be changed to support transactional behaviour. There are no changes to the interfaces of the POS to support transactions. The following discussion applies to support of persistence when a Transaction Service is required.

Support for persistence can be built from a number of other, more specialised, services that can also be shared by other object services. Examples include:

- Recovery Service: This supports the atomicity and durability properties.

- **Logging Service:** This is used by the Recovery Service to assist in supporting the atomicity and durability properties. It is also used by the Transaction Service to support the two-phase commit protocol.
- **Backup/Restore Service:** This supports the isolation property.

This view is consistent with the X/Open Distributed TP Model which separates the transaction manager service (that is, the implementation of a generalised two-phase commit protocol) from a Resource Manager which provides the set of services for data which has a life beyond process execution. This permits both transactions on transient objects, and persistent objects without transactions, each of which is desirable.

2.5.8 Concurrency Control Service

The Concurrency Control Service does not depend on any other service as such. Nevertheless, it is designed to work with the Transaction Service.

2.6 Relationship to CORBA

This section provides information about the relationship of other services to the CORBA Specification.

2.6.1 ORB Interoperability Considerations

Naming Service

Entities that are not CORBA objects — that is, not objects accessed via an Object Request Broker — are used for names (in the guise of pseudo-objects). In both cases, the interfaces to these entities conform as closely as possible to OMG IDL while satisfying the specific service design requirements, in order to enable maximum flexibility in the future. Specifically, in the Naming Service, name objects are pseudo-objects with interfaces defined in pseudo IDL (PIDL). These objects look like CORBA objects but are specifically designed to be accessed using a programming language binding. This is done for reasons based on the expected use of these objects.

Life Cycle Service

The Life Cycle Service assumes CORBA implementations support object relocation.

Persistent Object Service

The Persistent Object Service requires CORBA Interface Repositories.

Relationship Service

The Relationship Service requires CORBA Interface Repositories to support the ability to dynamically determine whether an **InterfaceDef** conforms to another **InterfaceDef**; that is, if it is a subtype. This is needed to implement type constraints for particular relationships.

Transaction Service

Some implementations of the Transaction Service will support:

- The ability of a single application to use both object and procedural interfaces to the Transaction Service. This is described as part of the specification, particularly in Section 7.4 on page 172 and Section 7.5 on page 185.
- The ability for different Transaction Service implementations to interoperate across a single ORB. This is provided as a consequence of this specification, which defines OMG IDL interfaces for all interactions between Transaction Service implementations.
- The ability for the same Transaction Service to interoperate with another instance of itself across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)
- The ability for different Transaction Service implementations to interoperate across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)
- A critical dependency for Transaction Service interoperation across different ORBs is the handling of the **propagation_context** between ORBs. This includes the following:
 - efficient transformation between different ORB representations of the **propagation_context**

- the ability to carry the ID information (typically an X/Open XID) between interoperating ORBs
- the ability to do interposition to ensure efficient local execution of the `is_same_transaction()` operation.

2.7 Relationship to Object Model

All specifications contained in this document conform to the OMG Object Model. No additional components or profiles are required by any service.

2.8 Conformance to Existing Standards

In general, existing relevant standards do not have object-oriented interfaces nor are they structured in a form that is easily mapped to objects. These specifications have been influenced by existing standards, and services have been designed which minimise the difficulty of encapsulating supporting software. The Naming Service specification is believed to be compatible with X.500, DCE CDS and ONC NIS and NIS+.

These specifications are broadly conformant to emerging ODP standards:

- ITU-T X.900 Series, ISO/IEC 10746, Reference Model — Open Distributed Processing
- ITU-T X.Trader, ISO/IEC 13235, Open Distributed Processing Trader.

Persistent Object Service Specification

3.1 Introduction

The goal of the Persistent Object Service (POS) is to provide common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The Persistent Object Service will be used in conjunction with other object services; for example, Naming, Relationship, Transaction, Life Cycle, and so on. The Persistent Object Service has the primary responsibility for storing the persistent state of objects, with other services providing other capabilities.

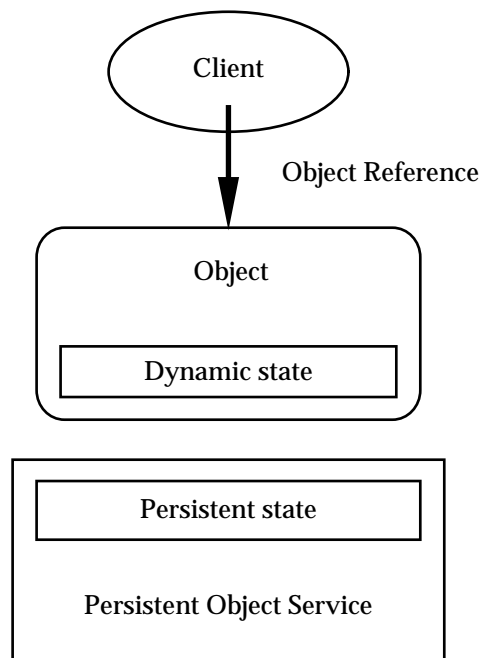


Figure 3-1 Roles in the Persistent Object Service

Figure 3-1 shows the participants in the Persistent Object Service. The state of the object can be considered in two parts: the *dynamic* state, which is typically in memory and is not likely to exist for the whole lifetime of the object (for example, it would not be preserved in the event of a system failure), and the *persistent* state, which the object could use to reconstruct the dynamic state.

Although the ORB provides the ability for an object reference to be persistent, it cannot ensure that the state of the object will be available just because the object reference is still valid.

The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. There is no requirement that any object use any particular persistence mechanism. For example, it may write its data to files using non-CORBA interfaces, or a single-level-store mechanism may be used. However, the Persistent Object Service provides capabilities that should be useful to a wide variety of objects.

Whether or not the client of an object is aware of the persistent state is a choice the object has.

CORBA already provides a persistent reference handling interface (that is, **object_to_string**, **string_to_object**, **release**, and so on). We expect that this will be sufficient for most clients to manage persistence of their referenced objects. But, because certain kinds of flexibility require the client to manage reference objects' persistence, the Persistent Object Service defines object interfaces for doing so. If this flexibility is not required, then these interfaces need not be supported or used.

The size, structure, access patterns and other properties of the dynamic and persistent state of the object varies tremendously. For many objects, their primary semantics are the efficient storage and access of their state for particular purposes. It is critical that the Persistent Object Service is able to support greatly different styles of usage and implementation in order to be useful to as many objects as possible.

As usual for object services, the primary task of this persistence specification is to define the interfaces that are needed to use the Persistent Object Service, and the conventions for how objects can work together using it.

The architecture of the Persistent Object Service defines multiple components and interfaces. In a particular situation, different parts of the service may be used. In no case does this specification assume the use of a particular implementation of a component, and it is expected that different implementations of the components will in fact work together.

Section 3.2 describes the overall goals and properties of the Persistent Object Service.

Section 3.3 defines the components which compose it.

Section 3.4 presents the **CosPersistencePID** module which defines the Persistence Identifier (PID).

Section 3.5 presents the **CosPersistencePO** module with interfaces borne by Persistent Objects.

Section 3.6 presents the interface to the Persistent Object Manager (POM).

Section 3.7 presents an overview of the Persistent Data Service (PDS) which interfaces both to the Protocol which communicates between **PO** and **PDS**, and to the Datastore which actually stores the data; following this, Section 3.8 on page 39 defines the **CosPersistencePDS** module which defines base functionality inherited by every protocol. Three protocols are presented in this specification, although more are possible:

- The Direct Access Protocol (PDS_DA) is described in Section 3.9, and its OMG IDL module is presented in Section 3.10.
- The ODMG-93 Protocol is described in Section 3.11.
- The Dynamic Data Object (DDO) Protocol is described in Section 3.12, and its OMG IDL module is presented Section 3.13. Other possible protocols are discussed briefly in Section 3.14. One possible datastore, implementable using a number of database and file mechanisms, is described in Section 3.15; other possible datastores are discussed in Section 3.16.

3.2 Goals and Properties

The Persistent Object Service plays a key role in structuring the object system. The model of how many objects work is critically dependent on consistent and integrated use of persistence. Like other object services, the Persistent Object Service provides interfaces that can support different implementations in order to obtain different qualities of service. Those interfaces allow different components to work together.

The overall persistence architecture has multiple components. Each will be introduced in turn in this section, following presentation of some basic capabilities and properties provided by the overall architecture.

3.2.1 Basic Capabilities

The principle requirement to be supported is for an object to be able to make all or part of its state persistent. Although CORBA defines object references as persistent (that is, they are usable until they are released regardless of the lifetime of their containing address space), it defines no particular way for the object to make its state persistent. The Persistent Object Service is intended ultimately to be the most common way to implement this. Therefore, there must be a way for the object to decide what state needs to be made persistent, and ways to store and retrieve that state.

It is often necessary to expose the persistent state from an object, so that the client can control the object's persistence to achieve certain types of flexibility. The Persistent Object Service defines a convention for doing this. Clients of objects sometimes need ways to refer to the persistent state, and request various operations on it. It is often not necessary to expose the persistent state from an object, so that the object implementation itself determines its persistence. In these cases, no persistence-specific object interfaces need be supported.

3.2.2 Object-oriented Storage

In existing non-object-oriented systems, persistence is accomplished by a number of data storage mechanisms. Generally, such mechanisms do not provide the key properties that object systems provide — uniform interfaces, self-description and abstraction. The Persistent Object Service brings these properties to storage by applying object technology and principles.

Interfaces to Data

To manage object persistence, the POS defines an architecture with interfaces defined using the CORBA IDL type system. Whether detailing the particular data to be stored, describing the protocol for accessing the state, or defining the convention for making state visible for client control, the same language is used. This makes persistence a natural part of the software environment. These interfaces are designed to be used in a wide variety of situations, creating uniformity by encouraging most objects to support them, while allowing optimisation and evolution.

By accessing data through an interface, many problems of data manipulation and exchange can be avoided. For example, programs always see data in the representation that is appropriate for the machine, programming language, and so on, of the application. Data can be translated as needed to facilitate use in different object types and implementations and for different storage formats or underlying persistent storage mechanisms (for example, stream files, record files or various databases) when it is accessed through the interface.

Self-description

A powerful characteristic of object-oriented systems is that the elements are self-describing. It is possible to determine from an object what kind of object it is and what interfaces it supports. In the persistence architecture this means, for example, that a client can determine whether or not an object wishes to make its persistent state visible by checking to see if the object supports the interface for doing so.

It also means that the data can be manipulated to some degree independently of the objects whose state they represent. This can allow generic facilities such as backup, migration, storage accounting, and so on, to be done independently of the objects whose state is being stored.

Abstraction

In order to support a wide and evolving set of uses, a service must be able to improve and replace its implementations without affecting the clients of that service. The desire for reuse of objects requires that those objects do not depend too strictly on other objects and services, but rather are willing to work with any other components that support the required interface.

A variety of value-added products are also possible assuming that the objects depend only on the defined interfaces. By interposing unexpected implementations, for example, it may be possible to support features such as replication or versioning in a transparent way.

3.2.3 Open Architecture

A major feature of the Persistent Object Service (and the OMG Object Services Architecture) is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where the mechanisms that are useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers may not be appropriate for mainframes.

Implementations can be lightweight, consisting of mostly library code, or powerful, leveraging decades of experience with database systems. Of course, the architecture specifies several interfaces, but also shows how new interfaces can be introduced when needed while still exploiting the rest of the architecture.

As with other object services, the Persistent Object Service is intended to be part of a collection of services. As a result, it does not attempt to solve all problems that might relate to storage. Rather, it assumes other services will provide the solutions. For example, the Persistent Object Service does not do naming, but assumes that the Naming Service will perform that function; it does not do transactions, but assumes that they will be added as appropriate; it does not handle issues of general compound objects, but assumes that there will be a scheme that spans Persistent Object, Life Cycle, Print and other services.

A key idea in object systems that is critical for persistence is the ability for new and existing storage services to be able to integrate into the architecture. The requirement for such components to “plug-and-play” together is paramount, since one cannot expect all data to be maintained in a particular kind of file or database system. Thus, the architecture has features to allow existing databases or other storage mechanisms to be used for persistence, and for new storage mechanisms to be developed that can support both Persistent Object Service clients and other kinds of clients.

The POS architecture is open with respect to **PersistentDataService**, **Datastore**, **Protocol** and **PID** interfaces. Although we define some minimum requirements for these in some cases, many alternatives are allowed, including ones that have not yet been defined.

3.2.4 Views of Service

There are multiple views of the service, and each participant may need to consider only a part of the architecture.

Client

It is common for clients of objects to need to control or to assist in managing persistence. In particular, the timing of when the persistent state is preserved or restored, and the identification of which persistent state is to be used for an object, are two aspects often of interest to clients. The ability of a client to see the object and its data separately allows different object implementations to be used with the same data and allows different files or databases and formats to be used with the same object implementation.

However, the client need only deal with such complexity when this type of functionality is necessary. The client of the object can be completely ignorant of the persistence mechanism, if the object chooses to hide it.

The Persistent Object Service provides an interface for objects to use when they want to expose their persistence to their clients. The interface does not completely abandon encapsulation, but gives the client visibility to those functions it needs. In fact, the client is generally unaware of how or if the object uses other parts of the Persistent Object Service.

Object Implementation

The object has the most involvement with persistence, and the most options in deciding how to use it. Defining and manipulating the persistent state of the object is often the most crucial part of its implementation. The first decision the object makes is what interface to its data it needs. The Persistent Object Service captures that choice in the selection of the Protocol used by the object. Some Protocols provide simple interfaces and limited functionality; others may provide more control and more powerful operations.

The object also has the choice of delegating the management of its persistent data to other services, or maintaining fine-grained control over it. The Persistent Object Service defines a Persistent Object Manager that handles much of the complexity of establishing connections between objects and storage, allowing new components to be introduced without affecting the objects or their clients.

The object may also provide the ability for its clients to manipulate its persistent state in various ways. This is important for creating a uniform view of persistence in the system.

Persistent Data Service

The Persistent Data Service (PDS) actually implements the mechanism for making data persistent and manipulating it. A particular PDS supports a Protocol defining the way data is moved in and out of the object, and an interface to an underlying Datastore.

The PDS has the responsibility of translating from the object world above it to the storage world below it. It plays critical roles in identifying the storage as well as providing convenient and efficient access to it.

We define multiple kinds of PDSs, each tuned to a particular protocol and data storage mechanism, since the range of requirements for performance, cost and qualitative features is so large. Multiple PDSs must work together to create the impression of a uniform persistence mechanism. The Persistent Object Manager provides the framework for PDSs to cooperate this way.

Datastore

The lowest-level interface we define is a Datastore. Although Datastore interfaces are the least visible part of the persistence architecture, they may be the most valuable, since there are so many different Datastores offering a wide spectrum of trade-offs between availability, data integrity, resource consumption, performance and cost, and it is expected that more will be created. By having an interface that is hidden from objects and their clients, a Datastore can provide service to any and all objects that indirectly use the Datastore interface.

The Datastore plays a key role in interoperating with other storage services. It is the manifestation in the object world of the various means of storing data that are not objects. Generally, standards for Datastore interfaces have already been defined for different kinds of data repositories — relational, object-oriented and file systems.

3.3 Service Structure

This section presents an overview of each of the major components and how they interrelate. Subsequent sections present the OMG IDL as divided into modules which correspond closely (but not exactly) to these components.

The major components of the Persistent Object Service are illustrated in Figure 3-1 on page 21. They are:

- Persistent Identifier (PID)

This describes the location of an object's persistent data in some Datastore and generates a string identifier for that data.

- Persistent Object (PO)

This is an object whose persistence is controlled externally by its clients.

- Persistent Object Manager (POM)

This component provides a uniform interface for the implementation of an object's persistence operations. An object has a single POM to which it routes its high-level persistence operations to achieve plug-and-play.

- Persistent Data Service (PDS)

This component provides a uniform interface for any combination of Datastore and Protocol, and coordinates the basic persistence operations for a single object.

- Protocol

This component provides one of several ways to get data in and out of an object.

- Datastore

This component provides one of several ways to store an object's data independently of the address space containing the object.

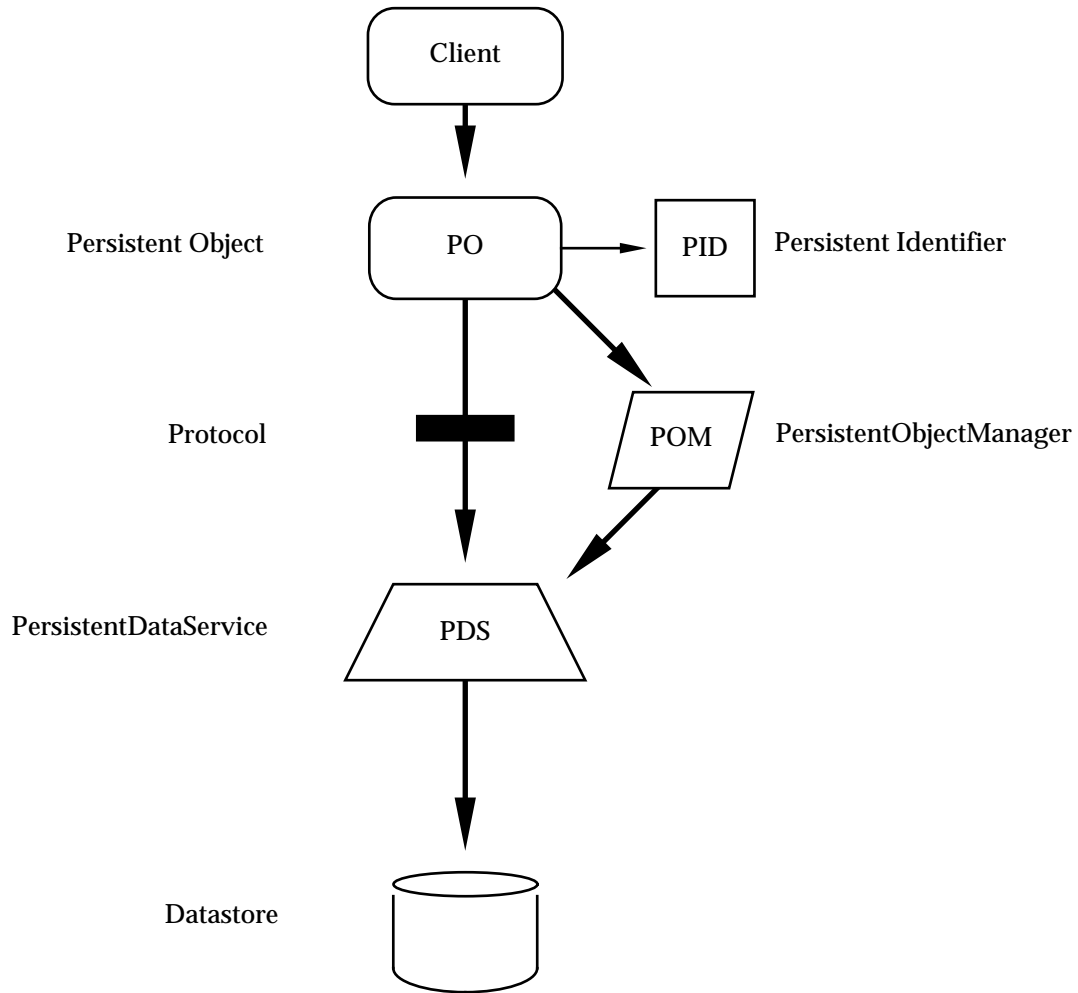


Figure 3-2 Major Components of the POS and their Interactions

The term *persistent object* is used to refer to objects whose persistence is controlled internally or externally. Either kind of persistent object can be supported by the Persistent Object Service's **POM**, **PDS**, **Protocol** and **Datastore** interfaces. The **PO** interface supports externally controlled persistence.

3.4 The CosPersistencePID Module

The **CosPersistencePID** module contains the basic interface for retrieving a PID: the **PID** interface. This section describes this interface, plus an example factory interface, and their operations in detail.

The **CosPersistencePID** module is shown below:

```
module CosPersistencePID {
    interface PID {
        attribute string datastore_type;
        string get_PIDString();
    };
};
```

The PID identifies one or more locations within a Datastore that represent the persistent data of an object and generates a string identifier for that data. An object must have a PID in order to store its data persistently. The client can create a PID, initialize its attributes, and connect it to the object. A persistent object's implementation uses the **POM** interface by passing the object and the PID as parameters.

The PID should not be confused with the CORBA object reference (OID). They are similar in that both have an operation that produces a string form that can be stored or communicated in whatever ways strings may be manipulated, and later used to get the original PID or OID. They differ in that the PID identifies data while the OID identifies a CORBA object.

For example, assume **mySpreadSheet** object is referenced by both **myDoc** and **yourDoc** objects. If **mySpreadSheet**'s OID is stored persistently with **myDoc** and **yourDoc** and then all three are brought into memory, both documents will always see the same spreadsheet object. If **mySpreadSheet**'s PID is stored persistently with **myDoc** and **yourDoc** and then all three object are brought into memory, each document will see a different spreadsheet object whose states will be the same initially but will diverge over time.

3.4.1 The PID Interface

The OMG IDL definition for the **PID** interface is as follows:

```
interface PID {
    attribute string datastore_type;
    string get_PIDString();
};
```

The **PID** interface contains at least one attribute:

```
attribute string datastore_type;
```

This identifies the interface of a Datastore. Example **datastore_types** might be **DB2**, **PosixFS** and **ObjectStore**. The PDS hides the Datastore's interface from the client, the persistent object and the POM, but PDS implementations are dependent on the Datastore's interface.

Other attributes can be added via subtyping the PID base type to reflect more specialised PIDs. Unless the **datastore_type** contains only a single object's persistent data, there is a need for more specific location information in the PID. The following example PID subtypes illustrate this:

```
#include "CosPersistencePID.idl"

interface PID_DB : CosPersistencePID::PID {
    attribute string database_name; // name of a database
};
```

```

interface PID_SQLDB : PID_DB {
    attribute string sql_statement; // SQL statement
};

interface PID_OODB : PID_DB {
    attribute string segment_name; // segment within database
    attribute unsigned long oid; // object id within a segment
};

```

The **PID** interface provides a single operation:

```
string get_PIDString( );
```

This operation returns a string version of the PID called the **PIDString**. A client should only obtain the **PIDString** using the **get_PIDString** operation. This allows the PID implementation to decide the form of the **PIDString**.

Some implementations may simply concatenate the PID attributes. Others may return a more compact form specialised for specific Datastores or even databases within a Datastore. Still others may return a universally unique identifier (UUID) that facilitates movement of its persistent data either within a single Datastore or between Datastores. A UUID-based PID might be implemented by overriding the **get** and **set** attribute operations and the **get_PIDString()** operation to bind and lookup the mapping between UUID and location information in a special context in the Naming Service. Using such a UUID-based PID, when an object is moved, the new location would be changed by setting the attributes to indicate the new location, and the PID would make the modification in the Naming Service. The **PIDString** would contain the UUID that does not change when an object's data is moved, so that references remain intact.

Some applications need to be able to restore an object given a PID but without knowing which type or implementation to use. The PID can be subtyped to accommodate this by adding the type or implementation as a PID attribute.

3.4.2 Example PIDFactory Interface

The OMG IDL definition for an example **PIDFactory** is as follows (others are also possible):

```

interface PIDFactory {
    CosPersistencePID::PID create_PID_from_key(in string key);
    CosPersistencePID::PID create_PID_from_string(
        in string pid_string);
    CosPersistencePID::PID create_PID_from_string_and_key(
        in string pid_string, in string key);
};

```

This example **PIDFactory** provides three ways of creating a PID:

- **CosPersistencePID::PID create_PID_from_key(in string key);**
This creates an instance of a PID given a key that identifies a particular PID implementation.
- **CosPersistencePID::PID create_PID_from_string(in string pid_string);**
This creates an instance of a PID given a **PIDString**. The **PIDString** must include some way to identify a particular PID implementation (the PID's key) that allows this operation to extract the PID's key from the **PIDString**. This key identifies the PID implementation for the newly created PID.
- **CosPersistencePID::PID create_PID_from_string_and_key(in string id_string, in string key);**
This creates an instance of a PID whose implementation is identified by the key in the input parameter instead of the key in the **PIDString**, and whose value is determined by the

PIDString. This is useful for when persistent data is moved between Datastores that require different **PID** interfaces.

3.5 The CosPersistencePO Module

The **CosPersistencePO** module collects the interfaces which are borne by a persistent object to allow its clients and the POM to control the PO's relationship with its persistent data. This module includes two interfaces:

- the **PO** interface
- the **SD** interface

plus an example factory interface.

The **PO** interface is borne by the PO and used by the client. The **SD** interface is borne by the PO and used by the POM.

This section describes these interfaces and their operations in detail.

The **CosPersistencePO** module is shown below:

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl
module CosPersistencePO {
    interface PO {
        attribute CosPersistencePID::PID p;
        CosPersistencePDS::PDS connect (
            in CosPersistencePID::PID p);
        void disconnect (in CosPersistencePID::PID p);
        void store (in CosPersistencePID::PID p);
        void restore (in CosPersistencePID::PID p);
        void delete (in CosPersistencePID::PID p);
    };
    interface SD {
        void pre_store();
        void post_restore();
    };
};
```

3.5.1 The PO Interface

The **PO** interface provides two mechanisms for allowing a client to externally control the PO's relationship with its persistent data:

- | | |
|---------------|--|
| Connection | This mechanism establishes a close relationship between the PO and its Datastore where the two data representations can be viewed as one for the duration of the connection. When the connection is ended, the data is the same in the PO and the Datastore, and the relationship between them no longer exists. An object can have only one connection at a time. |
| Store/Restore | These operations allow the client to move data between the PO and its Datastore in each direction separately, with each movement in each direction explicitly initiated by the client. |

The **PO** interface operations allow client control of a single PO's persistent data. When one of these operations is performed on a PO, the data included in these operations is up to that PO's implementation. For example, only part of the PO's private data may be included. Other POs may be included based on any criteria. If other POs are included, the target PO's implementation becomes their client and is responsible for controlling their persistence.

A PO client is responsible for the following:

- Creating a PID for the PO and initializing the PID. For storage, whatever location information is not specified will be determined by the Datastore. For a retrieval or delete operation, the location information must be complete.
- Controlling the relationship between the data in the PO and the Datastore. This is done by asking the PO to **connect()**, **disconnect()**, **store()**, **restore()** or **delete()** itself.

The OMG IDL definition for a PO is as follows:

```
interface PO {
    attribute CosPersistencePID::PID p;
    CosPersistencePDS::PDS connect (
        in CosPersistencePID::PID p);
    void disconnect (in CosPersistencePID::PID p);
    void store (in CosPersistencePID::PID p);
    void restore (in CosPersistencePID::PID p);
    void delete (in CosPersistencePID::PID p);
};
```

The PO interface has the following operations:

- **CosPersistencePDS::PDS connect (in CosPersistencePID::PID p);**

This begins a connection between the data in the PO and the Datastore location indicated by the PID. The persistent state may be updated as operations are performed on the object. This operation returns the PDS that handles persistence for use by those Protocols that require the PO to call the PDS.

- **void disconnect (in CosPersistencePID::PID p);**

This ends a connection between the data in the PO and the Datastore location indicated by the PID. It is undefined whether or not the object is usable if not connected to persistent state. The PID can be nil.

- **void store (in CosPersistencePID::PID p);**

This copies the persistent data out of the object in memory and puts it in the Datastore location indicated by the PID. The PID can be nil.

- **void restore (in CosPersistencePID::PID p);**

This copies the object's persistent data from the Datastore location indicated by the PID and inserts it into the object in memory. The PID can be nil.

- **void delete (in CosPersistencePID::PID p);**

This deletes the object's persistent data from the Datastore location indicated by the PID. The PID can be nil.

To adhere to the plug-and-play philosophy, objects pass these requests through to the POM, so that the interface for PO parallels that of the POM. This delegation to the POM allows objects to change PDSs (combination of Datastore and Protocol) without changing their implementation.

3.5.2 The POFactory Interface

The OMG IDL definition for an example **POFactory** is as follows (others are also possible):

```
#include "CosPersistencePO.idl"
// CosPersistencePO.idl #includes CosPersistencePDS.idl
// CosPersistencePDS.idl #includes CosPersistencePID.idl

interface POFactory {
    CosPersistencePO::PO create_PO (
        in CosPersistencePID::PID p,
        in string pom_id);
};
```

The example **POFactory** provides the following operation:

- **CosPersistencePO::PO create_PO(in CosPersistencePID::PID p, in string pom_id);**

This creates an instance of a PO that knows which POM to use and with its **pid** attribute already assigned.

3.5.3 The SD Interface

Some objects may be implemented knowing they are going to be persistent. Many such objects have both transient and persistent data. The Synchronized Data (SD) interface is provided to allow such objects to synchronise their transient and persistent data. Operations on the SD are invoked only by the POM. Persistent objects whose persistence is controlled either internally or externally (PO) can support the **SD** interface.

The OMG IDL definition for SD is as follows:

```
interface SD {
    void pre_store();
    void post_restore();
};
```

The interface for SD provides two operations:

- **void pre_store();**

This ensures that the persistent data is synchronised with the transient data.

- **void post_restore();**

This ensures that the transient data is synchronised with the persistent data.

A word processing document provides a good example of how these operations might be implemented. Suppose the document type is implemented with the following data:

- text buffer (persistent)
- attributes (persistent)
- text cache (transient)
- cursor location (transient).

The document could be implemented such that all work is done in the text cache. Then at store time, the text buffer needs to be updated, since it contains the actual data that will be stored. As such, the **pre_store()** operation should be implemented such that any updates in the text cache are propagated to the text buffer. The **post_restore()** operation should be implemented such that the text cache is initialized with a state consistent with the text buffer.

3.6 The CosPersistencePOM Module

The **CosPersistencePOM** module contains the interface which is borne by the POM and used by the PO. It contains a single interface: the **POM** interface.

This section describes this interface and its operations in detail.

The **CosPersistencePOM** module is shown below:

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePOM {
    interface Object;
    interface POM {
        CosPersistencePDS::PDS connect (
            in Object obj,
            in CosPersistencePID::PID p);
        void disconnect (
            in Object obj,
            in CosPersistencePID::PID p);
        void store (
            in Object obj,
            in CosPersistencePID::PID p);
        void restore (
            in Object obj,
            in CosPersistencePID::PID p);
        void delete (
            in Object obj,
            in CosPersistencePID::PID p);
    };
};
```

Clients of a PO will see the operations of the **POM** interface indirectly through the **PO** interface. The implementation of a persistent object with either externally or internally controlled persistence can use the **POM** interface. The POM provides a uniform interface across all PDSs, so different PDSs (combination of Datastore and Protocol) can be used without changing the object's implementation.

The OMG IDL definition of the POM is as follows:

```
interface POM {
    CosPersistencePDS::PDS connect (
        in Object obj,
        in CosPersistencePID::PID p);
    void disconnect (
        in Object obj,
        in CosPersistencePID::PID p);
    void store (
        in Object obj,
        in CosPersistencePID::PID p);
    void restore (
        in Object obj,
        in CosPersistencePID::PID p);
    void delete (
        in Object obj,
        in CosPersistencePID::PID p);
};
```

The **POM** interface has the following operations:

- **CosPersistencePDS::PDS connect (in Object obj, in CosPersistencePID::PID p);**
This begins a connection between data in the object and the Datastore location indicated by the PID. The persistent state may be updated as operations are performed on the object. This operation returns the PDS that is assigned the object's PID for use by those Protocols that require the PO to call the PDS.
- **void disconnect (in Object obj, in CosPersistencePID::PID p);**
This ends a connection between the data in the object and the Datastore location indicated by the PID. It is undefined whether or not the object is usable if not connected to persistent state. The PID can be nil.
- **void store (in Object obj, in CosPersistencePID::PID p);**
This gets the persistent data out of the object in memory and puts it in the Datastore location indicated by the PID. The PID can be nil.
- **void restore (in Object obj, in CosPersistencePID::PID p);**
This gets the object's persistent data from the Datastore location indicated by the PID and inserts it into the object in memory. The PID can be nil.
- **void delete (in Object obj, in CosPersistencePID::PID p);**
This deletes the object's persistent data from the Datastore location indicated by the PID. The PID can be nil.

The major function of the POM is to route requests to a PDS that can support the combination of Protocol and Datastore needed by the persistent object. To do this, the POM must know which PDSs are available and which Protocol and Datastore combinations they support. There are several possible ways that this information can be made available to a POM:

- How a Protocol is associated with an object.
One possibility is for the client to set the Protocol for that object. Another possibility is for the Protocol to be associated with the object's type or implementation.
- How a POM finds out the set of available PDSs and which Protocol (or object type) and Datastores they support.
One possibility is for the POM to find the information in a configuration file or a registry. Another possibility is to provide an interface to the POM for registering the information. The best or most natural technique may depend on the environment.

Because there are multiple ways to accomplish the above and more experience is needed to better understand whether there is a better way and what that might be, a **POM** interface for registering this information in the POM is not specified at this time.

When the POM is asked to store an object, the following steps logically occur:

- From the PID, the POM gets the **datastore_type** attribute.
- Regardless of how the Protocol is associated with the object, the POM uses the combination of Protocol and **datastore_type** to determine the PDS.
- The POM passes the store request through to the PDS.
- The PDS gets data from the object using a Protocol and stores the data in the Datastore.

The routing function of the POM serves to shield the client from having to know the details of how actual data storage/retrieval takes place. A client can change the repository of an object by changing the PID. The change will result in routing the next `store()` or `restore()` request to whatever the appropriate PDS is for the new Datastore.

Figure 3-3 on page 38 illustrates an example of the routing logic for the storage of `myDoc` in a DB2 database. This figure and the following example steps assume that, for this POM, the Protocol is associated with object type:

- The POM is asked to perform a store on `myDoc` with `pid1`.
- The POM finds the `datastore_type` associated with `pid1` (for example, DB2).
- The POM finds the object type of `myDoc` (for example, document).
- The POM determines that `myDoc` will use a particular PDS (for example, `pds1`).
- The POM routes the `store()` or `restore()` to `pds1`.
- The PDS gets the persistent data using `protocol1` and stores the data in the DB2 Datastore at `pid1`.

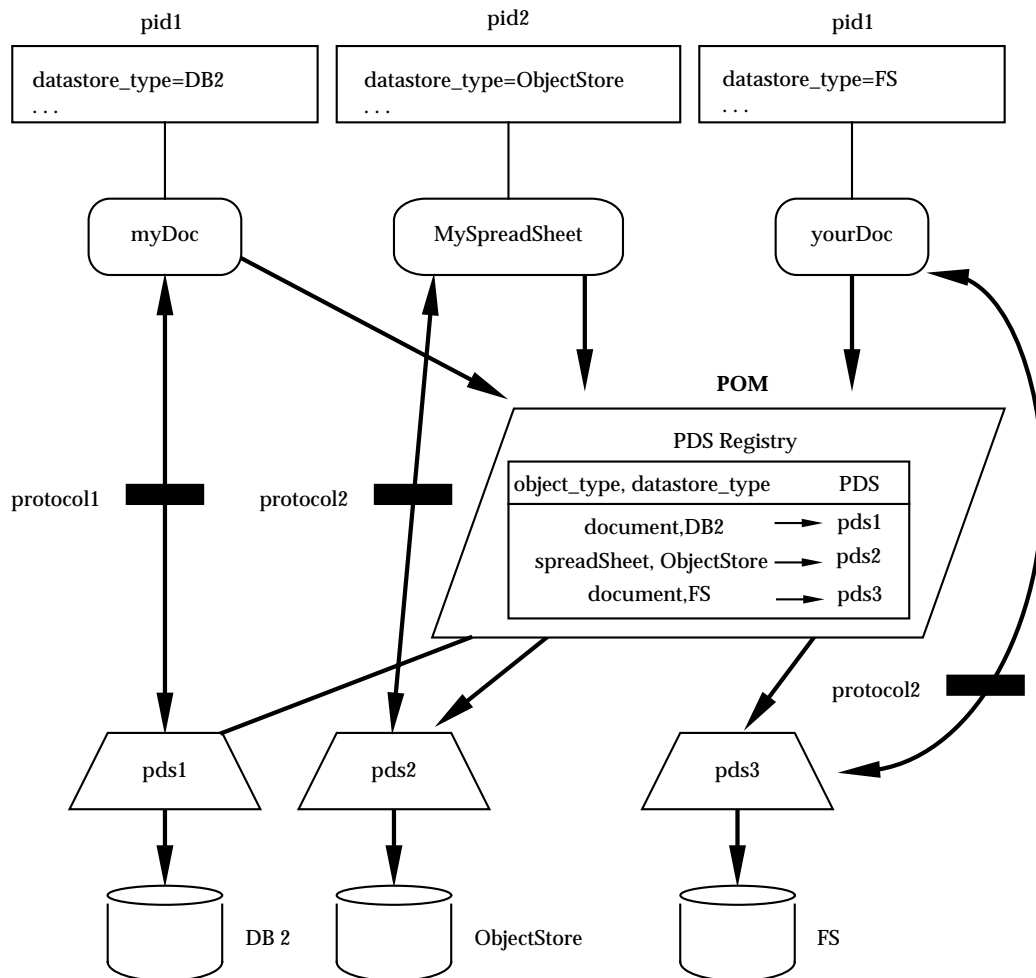


Figure 3-3 Example to Illustrate POM Functions

3.7 Persistent Data Service (PDS) Overview

The PDS implementation is responsible for the following:

- Interacting with the object to get data in and out of the object using a *Protocol*. Protocols are introduced in this section; three example Protocols and a discussion of additional Protocols are presented in Section 3.9 through Section 3.14.
- Interacting with the Datastore to get data in and out of the object. Datastores are introduced in this section, and an example Datastore plus a discussion of implementing additional Datastores are presented in Section 3.15 and Section 3.16.

A PDS performs the work for moving data into and out of an object and moving data into and out of a Datastore. There can be a wide variety of implementations of PDSs which provide different performance, robustness, storage efficiency, storage format or other characteristics, and which are tuned to the size, structure, granularity or other properties of the object's state.

Because the range of storage requirements is so large, there may be different ways in which the object can best access its persistent data, and there may be different ways in which the PDS can store that data. The way in which the object interacts with the PDS is called the Protocol. A Protocol may consist of calls from the object to the PDS, calls from the PDS to the object, implicit operations implemented with hidden interfaces, or some combination. The interaction *might* be explicit (for example, asking the object to stream out its data), or *implicit* (for example, the object might be mapped into persistent virtual memory). The Protocol is initiated when an object's persistent state is stored, restored or connected; this may be initiated by a POM or by the object itself. What happens after that depends on the particular Protocol. An object that uses a particular Protocol can work with any PDS that supports that Protocol. There is no "standard" Protocol. This specification defines three protocols: the Direct Attribute (DA) Protocol, the ODMG Protocol and the Dynamic Data Object (DDO) Protocol. A PDS might also use a programming language-specific or run-time environment-specific or other Protocol.

A PDS may use either a standard or a proprietary interface to its Datastore. A Datastore might be a file, virtual memory, some kind of database, or anything that can store information. This specification defines one Datastore interface that can be implemented by a variety of databases.

The PDS component interface is specified here as one module containing only the base **PDS** interface, plus one additional module per Protocol. Each Protocol-specific module inherits from the base module, augmenting the base functionality as needed.

3.8 The CosPersistencePDS Module

The **CosPersistencePDS** module contains the base interface upon which protocol-specific interfaces are built. It contains a single interface: the **PDS** interface.

This section describes this interface and its operations in detail.

The **CosPersistencePDS** module is shown below.

Some protocols may require specialisation of the **PDS** interface. However, no matter what Protocol or Datastore is used, a PDS always supports at least the following interface:

```
#include "CosPersistencePID.idl"
```

```
module CosPersistencePDS {
    interface Object;
    interface PDS {
        PDS connect (in Object obj,
                    in CosPersistencePID::PID p);
        void disconnect (in Object obj,
                        in CosPersistencePID::PID p);
        void store (in Object obj,
                   in CosPersistencePID::PID p);
        void restore (in Object obj,
                     in CosPersistencePID::PID p);
        void delete (in Object obj,
                    in CosPersistencePID::PID p);
    };
};
```

The exact semantics of the **connect()**, **disconnect()**, **store()** and **restore()** operations depend on the Protocol, since there may be other steps involved in the Protocol. In all four operations, the persistent state is determined by the PID of the object.

- **PDS connect (in Object obj, in CosPersistencePID::PID p);**

This connects the object to its persistent state, after disconnecting any previous persistent state. The persistent state may be updated as operations are performed on the object.

- **void disconnect (in Object obj, in CosPersistencePID::PID p);**

This disconnects the object from the persistent state. It is undefined whether or not the object is usable if not connected to persistent state.

- **void store (in Object obj, in CosPersistencePID::PID p);**

This saves the object's persistent state.

- **void restore (in Object obj, in CosPersistencePID::PID p);**

This loads the object's persistent state. The persistent state will not be modified unless a store or other mutating operation is performed on the persistent state.

- **void delete (in Object obj, in CosPersistencePID::PID p);**

This disconnects the object from its persistent state and deletes the object's persistent data from the Datastore location indicated by the PID.

3.9 The Direct Access (PDS_DA) Protocol

The first Protocol to be described here is the **PDS_DA** or Direct Access Protocol. The Direct Access Protocol supports direct access to persistent data through typed attributes organised in data objects that are defined in a Data Definition Language (DDL). An object using this Protocol would represent its persistent data as one or more interconnected data objects. For uniformity, the persistent data of an object is described as a single data object; however, that data object might be the root of a graph of data objects interconnected by stored data object references. If an object uses multiple data objects, the object traverses the graph by following stored data object references.

An object must define the types of the data objects it uses. Those types are specified in DDL, which is a subset of the OMG Interface Definition Language (OMG IDL) in which objects consist solely of attributes. The state of the data object is accessed using the attribute access operations defined in the CORBA Specification in conjunction with the appropriate programming language mapping.

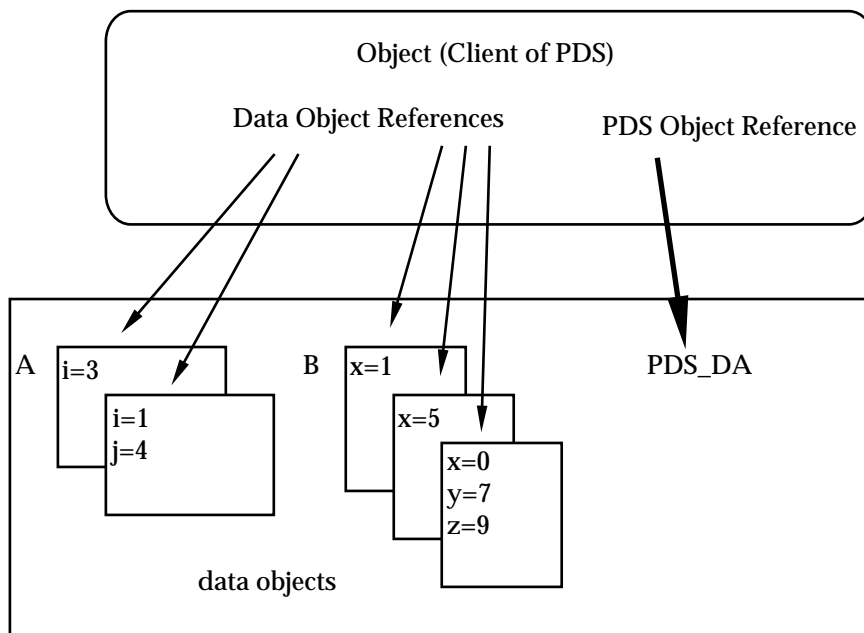


Figure 3-4 Direct Access Protocol Interfaces

The PDS_DA Protocol has two parts, as shown in Figure 3-4. When connected to a PDS, the object (which is effectively the client of the PDS) has an object representing the PDS which supports the **PDS_DA** interface. The object performs operations defined in the **PDS_DA** interface to get references to the data objects in the PDS. The persistent data is manipulated by performing operations using the data object references to get and set attributes on the collection of data objects in the PDS.

3.10 The CosPersistencePDS_DA Module

The `CosPersistencePDS_DA` module is a collection of interfaces which together define the Protocol. This module contains the following interfaces:

- the `PID_DA` interface
- the `DAObject` interface
- the `DAObjectFactory` interface
- the `DAObjectFactoryFinder` interface
- the `PDS_DA` interface
- the `DynamicAttributeAccess` interface
- the `PDSClustered_DA` interface.

This section describes these interfaces and their operations in detail.

The `CosPersistencePDS_DA` module is shown below.

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePDS_DA {

    typedef string DAObjectID;

    interface PID_DA : CosPersistencePID::PID {
        attribute DAObjectID oid;
    };
    interface DAObject {
        boolean dado_same(in DAObject d);
        DAObjectID dado_oid();
        PID_DA dado_pid();
        void dado_remove();
        void dado_free();
    };

    interface DAObjectFactory {
        DAObject create();
    };

    interface DAObjectFactoryFinder {
        DAObjectFactory find_factory(in string key);
    };

    interface PDS_DA : CosPersistencePDS::PDS {
        DAObject get_data();
        void set_data(in DAObject new_data);
        DAObject lookup(in DAObjectID id);
        PID_DA get_pid(); PID_DA get_object_pid(in DAObject dao);
        DAObjectFactoryFinder data_factories();
    };

    typedef sequence<string> AttributeNames;
    interface DynamicAttributeAccess {
        AttributeNames attribute_names();
        any attribute_get(in string name);
        void attribute_set(in string name, in any value);
    };
};
```

```

};

typedef string ClusterID;
typedef sequence<ClusterID> ClusterIDs;
interface PDS_ClusteredDA : PDS_DA{
    ClusterID cluster_id();
    string cluster_kind();
    ClusterIDs clusters_of();
    PDS_ClusteredDA create_cluster(in string kind);
    PDS_ClusteredDA open_cluster(in ClusterID cluster);
    PDS_ClusteredDA copy_cluster(
        in PDS_DA source);
};
};

```

3.10.1 The PID_DA Interface

The Persistent Identifiers (PIDs) used by the **PDS_DA** interface contain an object identifier that is local to the particular PDS. This value may be accessed with the following extension to the **CosPersistencePID** interface:

```

interface PID_DA : CosPersistencePID::PID {
    attribute DAObjectID oid;
};

```

The **DAObjectID** has the following attribute:

- **DAObjectID oid();**

This returns the data object identifier used by this PDS for the data object specified by the PID. The **DAObjectID** type is defined as an unbounded sequence of bytes that may be vendor-dependent.

3.10.2 The Generic DAObject Interface

The **DAObject** interface defined below provides operations that many data object clients need. A Datastore implementation may provide support for these operations automatically for its data objects. A data object is not required to support this interface. A client can obtain access to these operations by narrowing a data object reference to the **DAObject** interface:

```

interface DAObject {
    boolean dado_same(in DAObject d);
    DAObjectID dado_oid(); PID_DA dado_pid();
    void dado_remove();
    void dado_free();
};

```

The **DAObject** has the following operations:

- **boolean dado_same(in DAObject d);**

This returns true if the target data object and the parameter data object are the same data object. This operation can be used to test data object references for identity.

- **DataObjectID dado_oid();**

This returns the object identifier for the data object. The scope of data object identifiers is implementation-specific, but is not guaranteed to be global.

- **PID_DA dado_pid();**
This returns a PID_DA for the data object.
- **void dado_remove();**
This deletes the object from the persistent store and deletes the in-memory data object.
- **void dado_free();**
This informs the PDS that the data object is not required for the time being, and the PDS may move it back to persistent store. The data object must be preserved and must be brought back the next time it is referenced. This operation is only a hint and is provided to improve performance and resource usage.

3.10.3 The DAObjectFactory Interface

The scheme for factories follows the Life Cycle Service specification. The factory supports the following interface:

```
interface DAObjectFactory {
    DAObject create();
};
```

The **DAObjectFactory** has the following operation:

- **DAObjectFactory create();**
This creates a new data object in the PDS.

3.10.4 The DAObjectFactoryFinder Interface

This scheme for factories follows the Life Cycle Service specification. The factory finder supports the following interface:

```
interface DAObjectFactoryFinder {
    DAObjectFactory find_factory(in string key);
};
```

The **DAObjectFactoryFinder** has the following operation:

- **DAObjectFactoryFinder find_factory(in string key);**
This finds a factory for data objects as specified by the key.

3.10.5 The PDS_DA Interface

The DA Protocol uses an extended **PDS** interface called **PDS_DA**:

```
interface PDS_DA : CosPersistencePDS::PDS {
    DAObject get_data();
    void set_data(in DAObject new_data);
    DAObject lookup(in DAObjectID id);
    PID_DA get_pid(); PID_DA get_object_pid(in DAObject dao);
    DAObjectFactoryFinder data_factories();
};
```

The **PDS_DA** interface provides the following operations:

- **DAObject get_data();**
This returns the single root data object of the PDS.

- **void set_data(in DAObject new_data);**
This sets the single root data object
- **DAObject lookup(in DAObjectID id);**
This finds a data object by object id.
- **PID_DA get_pid();**
This constructs a PID that corresponds to the single root data object of this PDS.
- **PID_DA get_object_pid(in DAObject dao);**
This constructs a PID that corresponds to the specified data object, which must be in this PDS.
- **DAObjectFactoryFinder data_factories();**
This returns a factory finder. The factory finder will provide factories for the creation of new data objects within the PDS.

3.10.6 Defining and Using DA Data Objects

A **PDS_DA** implements data objects that have a set of attributes defined in a Data Definition Language (DDL). DDL is a subset of OMG IDL. In DDL, all interfaces consist only of attributes; that is, there are no operations. The programming interface for accessing the persistent state is the CORBA-defined attribute access operations as specified in the particular programming language mapping. A **PDS_DA** implements those accessor operations and transfers the persistent state between the Datastore and data objects as necessary.

DA data objects are used like normal CORBA objects. They are manipulated using object references, sometimes called data object references. Language mappings to data object interfaces are generated just like language mappings for other interfaces.

To define a DA data object (DADO), the developer decides what state must be made persistent. For example, suppose the object's persistent data consists of two values, one integer and one floating point number. The developer would define a data object interface **MyDataObject** describing this data:

```
interface MyDataObject {
    attribute short my_short;
    attribute float my_float;
};
```

The DDL definition must be compiled, installed and linked with the object implementation as necessary for the particular PDS and CORBA environment. Mechanisms similar to those for creating stubs for OMG IDL interfaces are used to provide the callable routines and create the run-time information necessary for the PDS implementation. The precise mechanisms are not defined in this specification.

Once the object has been connected to the PDS, the factory operations described above are used to create the data object and set it as the root object in the PDS. The object gets or sets values for the attributes using the CORBA accessor operations, for example:

```
// PDS_DA Examples
// C++ code
// Include OMG IDL compiler output from CosPersistencePDS_DA.idl
#include "CosPersistencePDS_DA.xh"
// CosPersistencePDS_DA.idl #includes CosPersistencePDS.idl
// CosPersistencePDS.idl #includes CosPersistencePID.idl
```

```

// connect to PDS
CosPersistencePDS_DA::PDS_DA my_pds =
    pom->connect(my_object,my_PID);
// get factory finder
DAObjectFactoryFinder daoff = my_pds->data_factories( );
// get factory for MyDataObject
DAObjectFactory my_factory =
    daoff->find_factory(MyDataObject);
// create an instance of MyDataObject
MyDataObjectRef my_obj =
    my_factory->create( );
// set the object to be the root object
my_pds->set_data(my_obj);
// put persistent state in attributes
my_obj->my_short(42);
my_obj->my_float(3.14159);
// use persistent state
my_obj->my_short(my_obj->my_short( )+12);

```

The DA Protocol allows developers to build simple object implementations that just read and write attribute values whenever they need to. There is no need for an object to cache persistent data in its transient store or to explicitly request it to be read or written.

Attributes can be defined using the full flexibility of the DDL type system. A particular PDS may restrict the attribute types it supports.

A data object may contain object references to other data objects and to ordinary CORBA objects. Here is an example that extends the previous example by adding a data object reference attribute and an ordinary CORBA object reference:

```

interface MyDataObject {
    attribute short my_short;
    attribute float my_float;
    attribute MyDataObject next_data;
    attribute SomeOtherObject my_object_ref; };

```

This example allows an instance of **MyDataObject** to refer to another instance. A Datastore implementation might restrict the scope of stored data object references. For example, it might permit only references to data objects in the same Datastore.

DDL interfaces support inheritance with semantics identical to OMG IDL. In the following example, a new type of data object is defined that has all the attributes of **MyDataObject**, plus an additional integer:

```

interface DerivedObject : MyDataObject {
    attribute short my_extra; };

```

Like other CORBA objects, data objects support operations on object references. In particular, the **get_interface()** operation, which returns an interface repository reference to the object's most derived interface, is useful for dynamically determining the type of a data object.

3.10.7 The DynamicAttributeAccess Interface

Because data objects are CORBA objects, the CORBA Dynamic Invocation interface can be used to get and set data object attributes dynamically, using strings to identify attributes at run time. However, to simplify dynamic access to data object attributes, the **DynamicAttributeAccess** interface is defined. This interface defines operations that allow determination of the names of the attributes of a data object and getting and setting individual attribute values by name. A data object is not required to support this interface. It can be determined whether or not a data object supports these operations by narrowing a data object reference to the **DynamicAttributeAccess** interface.

```
typedef sequence<string> AttributeNames;
interface DynamicAttributeAccess {
    AttributeNames attribute_names();
    any attribute_get(in string name);
    void attribute_set(in string name, in any value);
};
```

- **AttributeNames attribute_names();**
This returns a sequence containing the names of the object's attributes.
- **any attribute_get(in string name);**
This returns the value of the specified attribute.
- **void attribute_set(in string name, in any value);**
This sets the value of the named attribute to the value specified by the **any** parameter.

3.10.8 The PDS_ClusteredDA Interface

It is often useful to group data objects together within a PDS. Common reasons include locking, sharing, performance, and so on. The **PDS_ClusteredDA** is an extension to the **PDS_DA**. A non-clustered **PDS_DA** is effectively a single cluster.

Each cluster is represented as a distinct instance of the **PDS_ClusteredDA** interface, although they will typically all be implemented by the same service using the same Datastore.

In addition to supporting the normal **PDS_DA** interface, a clustered **PDS_DA** has the following interface:

```
typedef string ClusterID; typedef sequence<ClusterID> ClusterIDs;
interface PDS_ClusteredDA : PDS_DA {
    ClusterID cluster_id();
    string cluster_kind();
    ClusterIDs clusters_of();
    PDS_ClusteredDA create_cluster(in string kind);
    PDS_ClusteredDA open_cluster(in ClusterID cluster);
    PDS_ClusteredDA copy_cluster( in PDS_DA source); };
```

- **ClusterID cluster_id();**
This returns the **id** of this cluster.
- **string cluster_kind();**
This returns the kind of this cluster.
- **ClusterIDs clusters_of();**
This returns a sequence of ClusterIDs listing all of the clusters in this Datastore.

- **PDS_ClusteredDA create_cluster(in string kind);**

This creates a new cluster of the specified kind in this Datastore and returns a PDS_ClusteredDA instance to represent it.

- **PDS_ClusteredDA open_cluster(in ClusterID cluster);**

This opens an existing cluster that has the specified ClusterID.

- **PDS_ClusteredDA copy_cluster(inPDS_DA source);**

This creates a new cluster, loading its state from the specified cluster, which may be implemented in a different Datastore.

3.11 The ODMG-93 Protocol

A group of Object-Oriented Database Management System (ODBMS) vendors has recently endorsed and published a common ODBMS specification called ODMG-93. This specification defines an extended version of OMG IDL for defining ODBMS object types as well as programming language interfaces for object manipulation.

The ODMG-93 Protocol is similar to the **DA** Protocol, in that the object accesses attributes organised as data objects. The primary difference is that the ODMG-93 Protocol uses the Object Definition Language (ODL) defined in ODMG-93 instead of DDL, and it uses the programming language mapping defined for data objects specified in ODMG-93, rather than the CORBA IDL attribute operations.

If the ODMG-93 database object inherits the **PDS_DA** interface, then the database object can be used with the rest of this specification. Objects using the ODMG-93 Protocol would manipulate persistent data using the interfaces specified in ODMG-93.

Note that in addition to using the ODMG-93 interface as another protocol, it would be straightforward to implement the **DA** Protocol using an ODMG-93 ODBMS as a PDS. Since the **DA** Protocol is a subset of the functionality in ODMG-93, in most programming languages the language mapping for the DDL attributes would be a trivial layer on the ODMG-93 mapping. Using the ODMG-93 Protocol would fully exploit the capabilities of ODMG-93; using an ODMG-93 ODBMS to implement the **DA** Protocol captures those objects that use **DA** Protocol.

3.12 The Dynamic Data Object (DDO) Protocol

The **DDO** Protocol is a Datastore-neutral representation of an object's persistent data. Its purpose is to contain all of the data for a single object. Figure 3-5 on page 50 illustrates an example of a DDO. A DDO has a single PID, **object_type** and set of data items whose cardinality is **data_count**. Each piece of data has a **data_name**, **data_value** and a set of properties whose cardinality is **property_count**. Each property has a **property_name** and a property value.

Although any data can be stored in a DDO, the following example illustrates how it might map onto a row in a table:

DDO	a row
data_count	number of rows
data_item	column
data_name	column name
data_value	column value
property_count	number of column properties
property_name	for example, type or size
property_value	for example, character or 255.

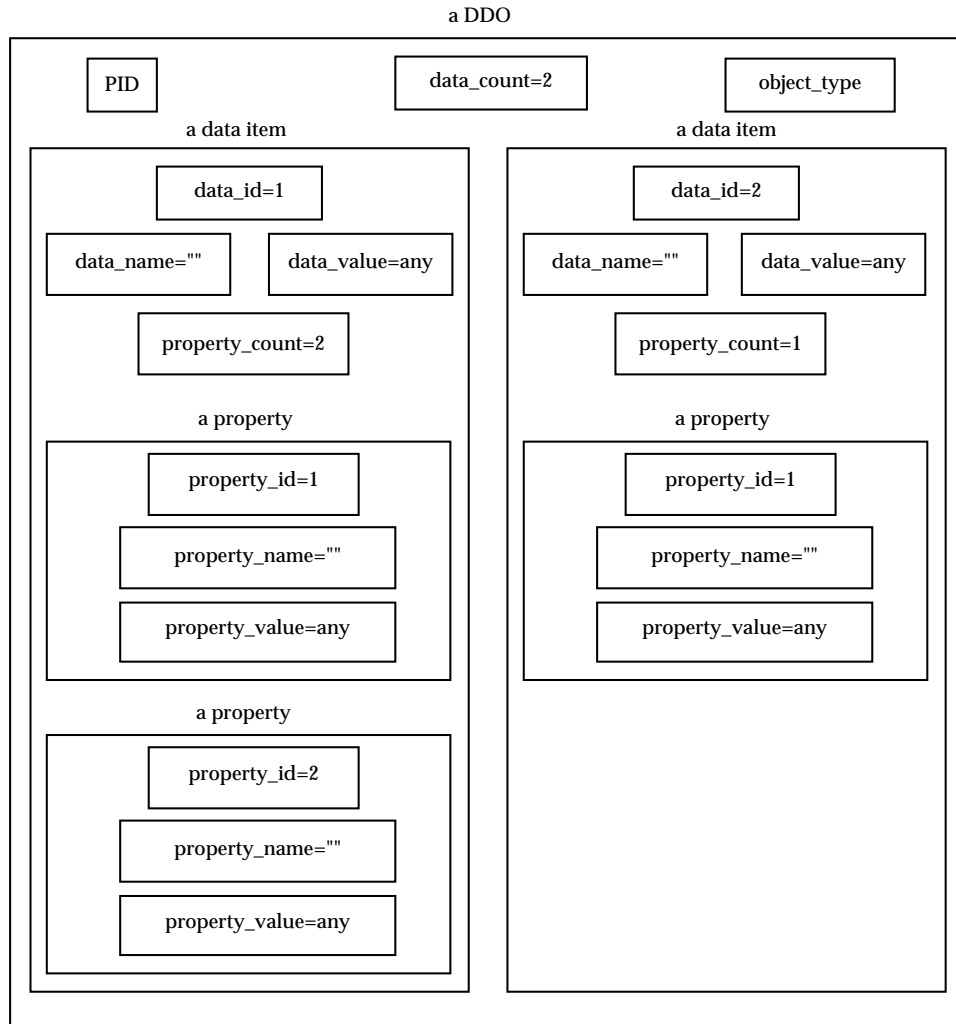


Figure 3-5 Structure of a DDO

A DDO provides a Protocol when the persistent object supports the **DDO** interface. In this case, the **DDO** interface is used to get data in and out of the persistent object. It may even provide the way that the persistent object stores its internal data, in which case a copy and reformat step is avoided.

To facilitate fast and simple storage and retrieval in specialised types of Datastore, DDOs can be used with particular conventions that are more suitable to different types of Datastore. If the DDO is used for both a Protocol and as a direct way to get data in and out of a Datastore, then copy and format costs are greatly reduced.

3.13 The CosPersistenceDDO Module

The **CosPersistenceDDO** module contains the OMG IDL to support the DDO protocol. The module contains one interface: the **DDO** interface.

This section describes the **CosPersistenceDDO** module in detail.

The **CosPersistenceDDO** module is shown below.

```
#include "CosPersistencePID.idl"

module CosPersistenceDDO {

    interface DDO {
        attribute string object_type;
        attribute CosPersistencePID::PID p;
        short add_data(); short add_data_property (in short data_id);
        short get_data_count();
        short get_data_property_count (in short data_id);
        void get_data_property (in short data_id,
            in short property_id,
            out string property_name,
            out any property_value);
        void set_data_property (in short data_id,
            in short property_id,
            in string property_name,
            in any property_value);
        void get_data (in short data_id,
            out string data_name,
            out any data_value);
        void set_data (in short data_id,
            in string data_name,
            in any data_value);
    };
};
```

A DDO has two attributes:

- **attribute string object_type;**
This identifies the **object_type** that this DDO is associated with.
- **attribute CosPersistencePID::PID p;**
This identifies the PID of the DDO.

A DDO has the following operations for getting data in and out of the DDO:

- **short add_data();**
This adds a new data item and returns a new **data_id** that can be used to access it.
- **short add_data_property (in short data_id);**
This adds a new property within the data item identified by **data_id** and returns the new **property_id** that can be used to access it within the context of the data item.
- **short get_data_count();**
This gets the number of data items in the DDO.

- **short get_data_property_count (in short data_id);**
This gets the number of properties associated with the data item identified by **data_id**.
- **void get_data_property (in short data_id, in short property_id, out string property_name, out any property_value);**
This gets the name and value of the property identified by **property_id** within the data item identified by **data_id**.
- **void set_data_property (in short data_id, in short property_id, in string property_name, in any property_value);**
This sets the name and value of the property identified by **property_id** within the data item identified by **data_id**.
- **void get_data (in short data_id, out string data_name, out any data_value);**
This gets the name and value of the data item identified by **data_id**.
- **void set_data (in short data_id, in string data_name, in any data_value);**
This sets the name and value of the data item identified by **data_id**.

3.14 Other Protocols

This specification includes three Protocols, but other Protocols can be supported in this architecture. The proliferation of Protocols would reduce the commonality of different objects, so it is desirable to use an existing Protocol if that is possible. However, when a new Protocol is required, it is still possible to use other parts of the Persistent Object Service with it. In general, the Protocol should be independent of the Datastore interface, although some Datastore interfaces will be better suited to some Protocols.

Some Protocols are already defined and are not specified here. Such standard interfaces as POSIX files are already in wide use, and there is no need to re-specify them. In this case, the PID would include the filename, and the Protocol would consist of reads and writes.

Other protocols are intended to be value-added and non-standard. For example, a LISP-specific PDS might take advantage of knowledge of the LISP run-time environment to create the appearance of a single-level store of LISP objects. Although such a PDS would not be usable from other programming languages, it could provide significant value to LISP programmers. Of course, it is also possible for a particular value-added protocol to be implemented as a layer on a standard protocol.

This specification allows such protocols to be integrated in the overall POS architecture without changing that architecture.

3.15 Datastores: The CosPersistenceDS_CLI Module

The last major component in the architecture is a Datastore, which provides operations on a data repository underneath the Protocols already discussed. As with Protocols, a variety of Datastore interfaces may be defined. There is no standard Datastore interface. Only one kind of Datastore is defined here, for record-oriented databases, because other standard interfaces already exist at this level and many customers may choose to omit this level of the architecture altogether for performance in an object-oriented database by using the DA or ODMG Protocol directly on the DBMS.

Datastore_CLI provides a uniform interface for accessing many different Datastores either individually or simultaneously. The acronym CLI refers to the X/Open CLI Specification on which the module is based. **Datastore_CLI** is especially suited for record database and file systems (for example, relational, IMS, hierarchical databases and VSAM file systems) that support user sessions, connections, transactions and scanning through data items using cursors.

The specification of this framework, where appropriate, is consistent with the X/Open CLI Specification, the IDAPI standard and the ODBC standard. These are industry standards which specify procedure-oriented application programming interfaces for accessing data stored in any type of Datastore.

More detailed explanations and enumeration of the options in the **Datastore_CLI** operations can be found in the X/Open CLI Specification.

DDOs are used as the way data is passed into the **Datastore_CLI** interface. If DDO is also being used as the Protocol, the PDS can use this DDO directly as a parameter to calls to the **Datastore_CLI**. When a different Protocol is being used, the PDS must create a new DO and populate it with data prior to calling the **Datastore_CLI**.

The **CosPersistenceDS_CLI** module contains the interfaces derived from the ODBC standard and the IDAPI standard, providing cursors into relational and other databases. The module contains the following interfaces:

- the **UserEnvironment** interface
- the **Connection** interface
- the **ConnectionFactory** interface
- the **Cursor** interface
- the **CursorFactory** interface
- the **PID_CLI** interface
- the **Datastore_CLI** interface.

This section describes these interfaces and their operations in detail.

The **CosPersistenceDS_CLI** module is shown below:

```
#include "CosPersistenceDDO.idl"
// CosPersistenceDDO.idl #includes CosPersistencePID.idl

module CosPersistenceDS_CLI {
    interface UserEnvironment {
        void set_option (in long option,in any value);
        void get_option (in long option,out any value);
        void release();
    };
};
```



```
interface Connection {
    void set_option (in long option,in any value);
    void get_option (in long option,out any value);
};

interface ConnectionFactory {
    Connection create_object (
        in UserEnvironment user_envir);
};

interface Cursor {
    void set_position (in long position,in any value);
    CosPersistenceDDO::DDO fetch_object();
};

interface CursorFactory {
    Cursor create_object (
        in Connection connection);
};

interface PID_CLI : CosPersistencePID::PID {
    attribute string datastore_id;
    attribute string id;
};

interface Datastore_CLI {
    void connect (in Connection connection,
        in string datastore_id,
        in string user_name,
        in string authentication);
    void disconnect (in Connection connection);
    Connection get_connection (
        in string datastore_id,
        in string user_name);
    void add_object (in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void delete_object (
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void update_object (
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void retrieve_object(
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    Cursor select_object(
        in Connection connection,
        in string key);
    void transact (in UserEnvironment user_envir,
        in short completion_type);
    void assign_PID (in PID_CLI p);
    void assign_PID_relative (
        in PID_CLI source_pid,
        in PID_CLI target_pid);
    boolean is_identical_PID (
        in PID_CLI pid_1,
        in PID_CLI pid_2);
    string get_object_type (in PID_CLI p);
    void register_mapping_schema (in string schema_file);
};
```

```

                Cursor execute (in Connection connection,
                               in string command);
        };
};

```

3.15.1 The UserEnvironment Interface

The **UserEnvironment** OMG IDL is as follows:

```

interface UserEnvironment {
    void set_option (in long option,in any value);
    void get_option (in long option,out any value); void release();
};

```

The **UserEnvironment** interface has the following operations:

- **void set_option (in long option, in any value);**
This sets the option to the desired value. The list of settable options is specified in the X/Open CLI Specification and the IDAPI standard.
- **void get_option (in long option, out any value);**
This gets the value of the option. The list of gettable options is the same as that for **set_option()**.
- **void release();**
This releases all resources associated with the **UserEnvironment**.

3.15.2 The Connection Interface

The **Connection** OMG IDL is as follows:

```

interface Connection {
    void set_option (in long option,in any value);
    void get_option (in long option,out any value);
};

```

The **Connection** interface contains the following operations:

- **void set_option (in long option, in any value);**
This sets the option to the desired value. The list of settable options is specified in the IDAPI standard.
- **void get_option (in long option, out any value);**
This gets the value of the option. The list of gettable options is the same as that for **set_option()**.

3.15.3 The ConnectionFactory Interface

The **ConnectionFactory** OMG IDL is as follows:

```

interface ConnectionFactory {
    Connection create_object (
        in UserEnvironment user_envir);
};

```

The **ConnectionFactory** interface has the following operation:

- **Connection create_object (in UserEnvironment user_envir);**

This creates an instance of **Connection**. A **Connection** is created within the context of a single **UserEnvironment**.

3.15.4 The Cursor Interface

The **Cursor** OMG IDL is as follows:

```
interface Cursor {
    void set_position (in long position,in any value);
    CosPersistenceDDO::DDO fetch_object();
};
```

A **Cursor** is a movable pointer into a list of DDOs, through which a client can move about the list or fetch a DDO from the list. The **Cursor** interface has the following operations:

- **void set_position (in long position, in any value);**

This sets the **Cursor** position to the desired value. The list of settable positions is specified in the IDAPI standard.

- **CosPersistenceDDO::DDO fetch_object();**

This fetches the next DDO from the list, based on the current position of the **Cursor**.

3.15.5 The CursorFactory Interface

The **CursorFactory** OMG IDL is as follows:

```
interface CursorFactory {
    Cursor create_object (
        in Connection connection);
};
```

The **CursorFactory** has the following operations:

- **Cursor create_object (in Connection);**

This creates an instance of **Cursor**. A **Cursor** is created within the context of a single **Connection**. See the X/Open CLI Specification and the IDAPI standard for more information.

3.15.6 The PID_CLI Interface

The **PID_CLI** OMG IDL is as follows:

```
interface PID_CLI : CosPersistencePID::PID {
    attribute string datastore_id;
    attribute string id; };
```

PID_CLI subtypes the **PID** base type (see Section 3.4.1 on page 29), adding attributes required for the **Datastore_CLI** interface. The **PID_CLI** interface has the following attributes:

- **attribute string datastore_id;**

This identifies the specific datastore in use. Most datastore products support multiple Datastores. For a relational database, this might be the name of a particular database containing multiple tables. For a POSIX file system, this might be the pathname of a file.

- **attribute string id;**

This identifies a particular data element within a Datastore. For a relational database, this might be a table name and primary key indicating a particular row in a table. For a POSIX

file system, this might be a logical offset within the file indicating where the data starts.

3.15.7 The Datastore_CLI Interface

The **Datastore_CLI** OMG IDL is as follows:

```
interface Datastore_CLI {
    void connect (in Connection connection,
                 in string datastore_id,
                 in string user_name,
                 in string authentication);
    void disconnect (in Connection connection);
    Connection get_connection (
        in string datastore_id,
        in string user_name);
    void add_object (in Connection connection,
                    in CosPersistenceDDO::DDO data_obj);
    void delete_object (
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void update_object (
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void retrieve_object(
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    Cursor select_object(
        in Connection connection,
        in string key);
    void transact (in UserEnvironment user_envir,
                  in short completion_type);
    void assign_PID (in PID_CLI p);
    void assign_PID_relative (
        in PID_CLI source_pid,
        in PID_CLI target_pid);
    boolean is_identical_PID (
        in PID_CLI pid_1,
        in PID_CLI pid_2);
    string get_object_type (in PID_CLI p);
    void register_mapping_schema (in string schema_file);
    Cursor execute (in Connection connection,
                   in string command);
};
```

In general, a client goes through the following steps to store, restore or delete DDOs:

1. Create a **UserEnvironment** and set the appropriate options to their desired values.
2. Create a **Connection** and set the appropriate options to their desired values. Open a connection to the Datastore, via **connect()**.
3. To store a DDO, call **add_object()** or **update_object()**. To restore a DDO, call **retrieve_object()**. To delete a DDO, call **delete_object()**.
4. If necessary, call **transact()** to commit or abort a Datastore transaction.
5. Repeat steps 3. and 4. as necessary.
6. Close the connection to the Datastore, via **disconnect()**. Delete the corresponding **Connection**.

7. Delete the **UserEnvironment**.

The **Datastore_CLI** connection operations are:

- **void connect (in connection connection, in string datastore_id, in string user_name, in string authentication);**

This opens a connection to the Datastore using the **Connection**. A client can establish more than one **Connection**, but only one **Connection** can be current at a time. The **Connection** that **connect()** establishes becomes the current **Connection**.

- **void disconnect (in Connection connection);**

This closes the **Connection**.

- **Connection get_connection (in string datastore_id, in string user_name);**

This returns the **Connection** associated with the **datastore_id**.

When any of the data manipulation operations is called, a Datastore transaction begins implicitly if the **Connection** involved is not already active. A **Connection** becomes active once the transaction begins and remains active until **transact()** is called.

The **Datastore_CLI** data manipulation operations are:

- **void add_object (in Connection connection, in CosPersistenceDDO::DDO data_obj);**

This adds the DDO to the Datastore. If necessary, get the mapping schema information for the DDO first.

- **void delete_object (in Connection connection, in CosPersistenceDDO::DDO data_obj);**

This deletes the DDO from the Datastore. If necessary, get the mapping schema information for the DDO first.

- **void update_object (in Connection connection, in CosPersistenceDDO::DDO data_obj);**

This updates the DDO in the Datastore. If necessary, get the mapping schema information for the DDO first.

- **void retrieve_object (in Connection connection, in CosPersistenceDDO::DDO data_obj);**

This retrieves the DDO from the Datastore. If necessary, get the mapping schema information for the DDO first. To improve performance, the **DBDatastore_CLI** may obtain access to more than one DDO at a time and cache these.

- **Cursor select_object (in Connection connection, in string key);**

This selects and retrieves the DDO(s) which match the key from the Datastore. The DDO(s) are returned through the **Cursor**. If necessary, get the mapping schema information for the key first. This operation is provided to support the Query Service. In addition, the **Datastore_CLI** will support any other operation required by the Query Service.

The **Datastore_CLI** functions as a Resource Manager for the DDOs that it manages. As such, it will support all Resource Manager operations specified by the Transaction Service. When the Transaction Service is not being used, a transaction is initiated implicitly by either a **Connection** or a **transact()**, and ended with a **transact()**.

- **void transact (in UserEnvironment user_envir, in short completion_type);**

This completes (commit or rollback) a Datastore transaction. Transaction completion enacts or undoes any **add_object()**, **update_object()** or **delete_object()** operations performed on any **Connection** within the **UserEnvironment** since the connection was established or since a

previous call to `transact()` for the same `UserEnvironment`. The values of `completion_type` are specified in the X/Open CLI Specification.

The `Datastore_CLI` PID operations are:

- **void assign_PID (in PID_CLI p);**

This assigns a value for the `id` attribute of the PID. The first attribute, `datastore_type`, must be filled in before calling this operation. If only the first attribute is filled in, then this operation will fill in the second attribute, `datastore_id`, as well.

- **void assign_PID_relative (in PID_CLI source_pid, in PID_CLI target_pid);**

This assigns values for the attributes of the `target_pid` based on the values of the `source_pid`. The `target_pid`'s first two attributes, `datastore_type` and `datastore_id`, will be assigned the same values as those of the `source_pid`. Its `id` attribute will be assigned a new value which is based on some relationship with that of the `source_pid`. The algorithm defining that relationship is up to the implementation.

- **boolean is_identical_PID (in PID_CLI pid_1, in PID_CLI pid_2);**

This tests to see if the two PIDs are identical. In order for the two PIDs to be identical, the following conditions must be true:

1. Both PIDs must be managed by this PDS.
2. All three attributes of the PIDs must be identical individually.

- **string get_object_type (in PID_CLI p);**

This gets the `object_type` of the PID.

Other `Datastore_CLI` operations are:

- **void register_mapping_schema (in string schema_file);**

This registers the mapping schema information contained within the `schema_file` with the `Datastore_CLI`. The mapping schema generally consist of individual mappings each of which is applicable to a given pair of `object_type` and `datastore_type`.

- **Cursor execute (in Connection connection, in string command);**

This executes a command on the Datastore. If there are any DDOs to be returned as a result, this is done through the `Cursor`.

3.16 Other Datastores

There are other Datastore interfaces that can be used by PDSs. Some of these interfaces are not CORBA object interfaces, in that they are not defined in OMG IDL and the Datastores are not objects.

Some Datastores are simple, such as POSIX files. Others may be databases, and may use generic interfaces for databases and record files such as SQL, the X/Open CLI Specification, the IDAPI standard and the ODBC standard. Some Datastores are tuned to support nested documents or other specific kinds of objects, such as Bento.¹

Because the Datastore interface is not exposed to object implementations or clients, the choice of Datastore interface is up to the PDS. As long as the PDS can support its Protocol using the particular Datastore interface, any implementation of the Datastore can be used by that PDS. The identification of data within different types of Datastores is facilitated by the PID, which can be specialised to each Datastore type.

3.17 Standards Conformance

This service is specified in standard OMG IDL.

The **Datastore_CLI** portion of the Persistent Object Service is consistent with the X/Open CLI Specification.

The ODMG-93 **PDS** object protocol incorporates the ODMG-93 specification.

1. Jed Harris and Ira Rubin, The Bento Specification, Revision 1.0d5, Apple Computer, Inc., July 1993.

Concurrency Control Service Specification

4.1 Service Description

The purpose of the Concurrency Control Service is to mediate concurrent access to an object such that the consistency of the object is not compromised when accessed by concurrently executing computations.

The Concurrency Control Service consists of multiple interfaces that support both transactional and non-transactional modes of operation. The user of the Concurrency Control Service can choose to acquire locks in one of two ways:

- On behalf of a transaction (transactional mode).

The Transaction Service drives the release of locks as the transaction commits or aborts.

- By acquiring locks on behalf of the current thread (that must be executing outside the scope of a transaction).

In this non-transactional mode, the responsibility for dropping locks at the appropriate time lies with the user of the Concurrency Control Service.

The Concurrency Control Service ensures that transactional and non-transactional clients are serialised. Hence a non-transactional client that attempts to acquire a lock (in a conflicting mode) on an object that is locked by a transactional client will block until the transactional client drops the lock.

4.1.1 Basic Concepts of Concurrency Control

Clients and Resources

The Concurrency Control Service enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

The Concurrency Control Service does not define what a resource is. It is up to the clients of the Concurrency Control Service to define resources and to properly identify potentially conflicting uses of those resources. In a typical use, an object would be a resource, and the object implementation would use the Concurrency Control Service to coordinate concurrent access to the object by multiple clients.

Transactions as Clients

The Concurrency Control Service differentiates between two types of client: a *transactional client* and a *non-transactional client*. Conflicting access by clients of different types is managed by the Concurrency Control Service, thereby ensuring that clients always see the resource in a consistent state.

The Concurrency Control Service does not define what a transaction is. Transactions are defined by the Transaction Service. The Concurrency Control Service is designed to be used with the Transaction Service to coordinate the activities of concurrent transactions.

The Transaction Service supports two modes of operation: *implicit* and *explicit*. When operating in the implicit mode, a transaction is implicitly associated with the current thread of control. When executing in the explicit mode, a transaction is specified explicitly by the reference to the coordinator that manages the current transaction. To simplify the model of locking supported by the Concurrency Control Service when a transactional client is operating in the implicit transaction mode, transactional clients are limited to a single thread per transaction (nested transactions can be used when parallelism is necessary) and that thread can be executing on behalf of at most one transaction at a time.

Locks

The Concurrency Control service coordinates concurrent use of a resource using locks. A lock represents the ability of a specific client to access a specific resource in a particular way. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the activities of those clients might conflict. To achieve coordination, a client must obtain an appropriate lock before accessing a shared resource.

Lock Modes

The Concurrency Control Service defines several lock modes, which correspond to different categories of access. Having a variety of lock modes allows more flexible conflict resolution. For example, providing different modes for reading and writing allows a resource to support multiple concurrent clients that are only reading the data of the resource. The Concurrency Control Service also defines intention locks that support locking at multiple levels of granularity.

Lock Granularity

The Concurrency Control Service does not define the granularity of the resources that are locked. It defines a lock set, which is a collection of locks associated with a single resource. It is up to clients of the Concurrency Control Service to associate a lock set with each resource. Typically, if an object is a resource, the object would internally create and retain a lock set. However, the mapping between objects and resources (and lock sets) is up to the object implementation; the mapping could be one-to-one, but it could also be one-to-many, many-to-many or many-to-one.

Conflict Resolution

A client obtains a lock on a resource using the Concurrency Control Service. The service will grant a lock to a client only if no other client holds a lock on the resource that would conflict with the intended access to the resource. The decision to grant a lock depends upon the modes of the locks held or requested. For example, a read lock conflicts with a write lock. If a write lock is held on a resource by one client, a read lock will not be granted to another client.

Conflict Resolution for Transactions

The decision to grant a lock also depends upon the relationships among the transactions that hold or request a lock. In particular, if the transactions are related by nesting (nested transactions), a lock may be granted that would otherwise be denied.

Lock Duration

Typically, a transaction will retain all of its locks until the transaction is completed (either committed or aborted). This policy supports serialisability of transactional operations. Using the two-phase commit protocol, locks held by a transaction are automatically dropped when the transaction completes.

There are also situations where levels of isolation that are weaker than serialisability are acceptable, such as when an application does not want other applications to change an object while reading it and does not refer to the object again within the transaction. In these circumstances, it is acceptable to release locks before the containing transaction completes, hence the duration will be shorter than the containing transaction.

To manage the release of the locks held by a transaction, the Concurrency Control Service defines a **LockCoordinator**. Lock sets that are related (for example, by being created by a Resource Manager for resources of the same type) and that should drop their locks together when a transaction commits or aborts, may share a **LockCoordinator**. It is up to clients of the Concurrency Control Service to associate lock sets together and to release the locks when a transaction commits or aborts.

4.2 Locking Model

This section covers a number of important issues that relate to the locking model supported by the Concurrency Control Service. For a complete discussion of these issues the reader is directed to one of the standard texts on the subject.²

Section 4.2.1 applies to clients that operate in both transactional and non-transactional modes. Section 4.2.2 on page 67, Section 4.3 on page 68 and Section 4.4 on page 69 are relevant only to clients that operate in transactional mode.

4.2.1 Lock Modes

Read, Write and Upgrade Locks

The Concurrency Control Service defines *read* (R) and *write* (W) lock modes that support the conventional multiple readers, one writer policy. Read locks conflict with write locks, and write locks conflict with other write locks.

In addition, the Concurrency Control Service defines an *upgrade* (U) mode. An upgrade mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock that occurs when two or more clients attempt to read and then update the same resource. If more than one client holds a read lock on the resource, a deadlock will occur as soon as one of the clients requests a write lock on the resource. If each client requests a single upgrade lock followed by a write lock, this deadlock will not occur.

Intention Read and Intention Write Locks

The granularity of the resources locked by an application determines the concurrency within the application. Coarse granularity locks incur low overhead (since there are fewer locks to manage) but reduce concurrency since conflicts are more likely to occur. Fine granularity locks improve concurrency but result in a higher locking overhead since more locks are requested. Selecting a suitable lock granularity is a balance between the lock overhead and the degree of concurrency required. Using the Concurrency Control service, an application can be developed to use coarse or fine granularity locks by defining the associated resources appropriately.

In addition, the Concurrency Control Service supports variable granularity locking using two additional lock modes, *intention read* (IR) and *intention write* (IW). These additional lock modes are used to exploit the natural hierarchical relationship between locks of different granularity.

For example, consider the hierarchical relationship inherent in a database: a database consists of a collection of files, with each file holding multiple records. To access a record, a coarse grain lock may be set on the database, but at the cost of restricting other clients from accessing the database. Clearly, this level of locking is unsuitable. However, only setting a lock on the record is also inappropriate, because another client might set a lock on the file holding the record and delete or modify the file.

Using variable granularity locking, a client first obtains intention locks on the ancestor(s) of the required resource. To read a record in the database; for example, the client obtains an intention read lock (IR) on the database and the file (in this order) before obtaining the read lock (R) on the record. Intention read locks (IR) conflict with write locks (W), and intention write locks (IW)

2. See the referenced *Concurrency Control and Recovery in Database Systems* or the referenced *Transaction Processing: Concepts and Techniques*.

conflict with read (R) and write (W) locks.

Lock Mode Compatibility

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

Table 4-1 Lock Compatibility

Table 4-1 defines the compatibility between the various locking modes (the symbol * is used to indicate when locks conflict). When a client requests a lock on a resource that cannot be granted because another client holds a lock on the resource in a conflicting mode, the client must wait until the holding client releases its lock. The Concurrency Control Service enforces a queueing policy such that all clients waiting for a new lock are serviced in a first in, first out order, and subsequent requests are blocked by the first request waiting to be granted the lock, unless the requesting client is a transaction that is a member of the same transaction family as an existing holder of the lock.

4.2.2 Multiple Possession Semantics

The Concurrency Control Service interface supports a locking model called *multiple possession semantics*. In this model, a client can hold multiple locks on the same resource simultaneously. The locks can be of different modes. In addition, a client can hold multiple locks of the same mode on the same resource; effectively, a count is kept of the number of locks of a given mode that have been granted to the client. When a client holds locks on a resource in more than one mode, other clients will not be granted a lock on the resource unless the requested lock mode is compatible with all of the modes of the existing locks.

In contrast, using the conventional locking model,³ when a client holding a lock on a resource requests a lock on the same resource in a stronger mode, the existing lock is promoted from the weaker mode to the stronger mode (once the stronger lock can be granted without causing a conflict). Since lock modes form only a partial order, there will not always be a stronger mode; in cases where neither mode is stronger, the lock will be promoted to the weakest mode that is at least as strong as either of the two modes.

3. See the referenced *Notes on Database Operating Systems*.

4.3 Two-phase Transactional Locking

The Concurrency Control Service provides primitives to support locking. Transaction duration locking is a special case of strict two-phase locking. In the first phase (the growing phase), a transaction obtains locks that are kept until the second phase (the shrinking phase), at which point they are released. A transaction must not release locks during the first phase, and must not obtain new locks during the second phase, otherwise concurrent computations may be able to view intermediate results of the transaction.

Two-phase locking is sufficient to guarantee serialisability, hence this technique is used by transactions. During the normal execution of a transaction, no locks will be automatically dropped before the end of the transaction. When the transaction completes, the Concurrency Control Service must be informed so that the locks the transaction holds may be released. While releasing locks, no new locks may be obtained by the transaction.

When a transaction holds a lock that is no longer needed to ensure the transaction or if a weaker level of isolation is acceptable, it is permissible to release the lock. The Concurrency Control Service therefore provides an operation that releases individual locks. This operation should be used with caution to ensure that the isolation level is appropriate for the application.

4.4 Nested Transactions

Lock conflicts within a transaction family are treated somewhat differently than conflicts between unrelated transactions. The underlying principle is the same for both: transactions must not be able to observe the effects of other transactions that might later abort. Unrelated transactions can abort independently; therefore, one transaction must not be permitted to acquire a lock that conflicts with a lock on the same resource held by an unrelated transaction.

Nesting imposes abort dependencies among related transactions. A parent transaction cannot abort without causing all of its children to abort. A child transaction that ends successfully cannot abort without causing its parent to abort. A transaction that cannot abort without causing another related transaction to abort (according to these guidelines and logical deductions) is said to be committed relative to that other transaction.

These dependencies make it possible to relax the rule that two transactions cannot acquire locks of conflicting modes on the same resource, without breaking the underlying principle. No partial effects can be observed and committed if all transactions that have done work cannot abort without the observer being aborted. This property translates into a simple rule for nested locking: if all transactions holding locks on a resource are committed with respect to a transaction trying to acquire a lock on the resource, no conflict exists.

The multiple possession model (see Section 4.2.2 on page 67) facilitates the use of locks with nested transactions. In this model, multiple related transactions may hold locks of conflicting modes on a resource at the same time. When a nested transaction requests a lock, it is granted if all of the transactions holding locks on the resource are committed relative to the requestor. Both the requestor and previous holders are then considered to hold locks on the resource.

A child transaction can acquire a lock on a resource locked by its parent and then drop that lock without causing its parent to lose its lock. A transaction cannot drop a lock that it did not acquire itself. The lock possession semantics also require that each transaction acquire locks on its own behalf. It is improper to take locks on behalf of another transaction or to depend on locks held by other transactions.

Other approaches to nested transactions⁴ treat a resource as being locked by a single transaction at a time. When a nested transaction requests a lock on a resource that is already locked by an ancestor transaction, the nested transaction becomes the new owner of the lock. When a nested transaction commits, ownership of all of its locks is transferred to its parent. When a nested transaction aborts, ownership of its locks reverts to the previous owners. The Concurrency Control Service performs these lock transfers automatically. The multiple possession semantics model is functionally equivalent to this model, but it supports simpler interfaces.

4. See the referenced *Nested Transactions: An Approach to Reliable Distributed Computing*.

4.5 The CosConcurrencyControl Module

The Concurrency Control Service is defined by the **CosConcurrencyControl** module, which provides interfaces that support both transactional and non-transactional modes of operation. This section defines the interfaces and describes the operations they support.

- The interfaces provide two modes of operation that correspond to those supported by the Transaction Service; in both modes, locks are identified by the lock set they are associated with and the mode of the lock.
- A client of the Concurrency Control Service may operate in an implicit mode such that locks are acquired on behalf of the current transaction (for transactional clients) or current thread (for non-transactional clients).
- For transactional clients, a second alternative is possible that involves the client identifying the transaction by means of a reference to the transaction's coordinator object (the explicit mode of operation).

Locks are acquired on lock sets. Two sets of operations are provided by the **LockSetFactory** interface to create lock sets. One creates a lock set that can be used by clients operating in the implicit mode (the **LockSet** interface), the other creates a lock set for explicit mode transactional clients (the **TransactionalLockSet** interface). In addition, the **LockCoordinator** interface is provided to allow a client to release all the locks held by a specific transaction.

The following sections define the types and exceptions common to both types of interface, the interfaces themselves. The responsibilities of a user for managing transaction-duration locks are also described.

OMG IDL for the **CosConcurrencyControl** module is shown below:

```
#include <CosTransactions.idl>
module CosConcurrencyControl {

    enum lock_mode {
        read,
        write,
        upgrade,
        intention_read,
        intention_write
    };
    exception LockNotHeld{};

    interface LockCoordinator
    {
        void drop_locks();
    };
    interface LockSet
    {
        void lock(in lock_mode mode);
        boolean try_lock(in lock_mode mode);

        void unlock(in lock_mode mode)
            raises(LockNotHeld);
        void change_mode(in lock_mode held_mode,
            in lock_mode new_mode)
            raises(LockNotHeld);
        LockCoordinator get_coordinator(
            in CosTransactions::Coordinator which);
    };
};
```



```

interface TransactionalLockSet
{
    void lock(in CosTransactions::Coordinator current,
              in lock_mode mode);
    boolean try_lock(in CosTransactions::Coordinator current,
                    in lock_mode mode);
    void unlock(in CosTransactions::Coordinator current,
               in lock_mode mode)
        raises(LockNotHeld);
    void change_mode(in CosTransactions::Coordinator current,
                    in lock_mode held_mode,
                    in lock_mode new_mode)
        raises(LockNotHeld);
    LockCoordinator get_coordinator(
        in CosTransactions::Coordinator which);
};
interface LockSetFactory
{
    LockSet create();
    LockSet create_related(in LockSet which);
    TransactionalLockSet create_transactional( );
    TransactionalLockSet create_transactional_related(
        in TransactionalLockSet which);
};
};

```

4.5.1 Types and Exceptions

The types and exceptions described in this section apply to both the **LockSet** and **TransactionalLockSet** interfaces.

```

module CosConcurrencyControl {
    enum lock_mode {
        read,
        write,
        upgrade,
        intention_read,
        intention_write
    };
};

```

```
exception LockNotHeld{};
```

- **lock_mode**

The **lock_mode** type represents the types of lock that can be acquired on a resource.

- **LockNotHeld**

The **LockNotHeld** exception is raised when an operation to unlock or change the mode of a lock is called and the specified lock is not held.

4.5.2 The LockCoordinator Interface

The **LockCoordinator** interface enables a transaction service to drop all locks held by a transaction. The **LockSet** and **TransactionalLockSet** interfaces create instances of the **LockCoordinator** for each transaction. The **LockCoordinator** interface provides a single operation:

```
interface LockCoordinator {
    void drop_locks();
};
```

- **drop_locks**

This releases all locks held by the transaction. This call is designed to be used by transactional clients when a transaction commits or aborts. For nested transactions, this operation must be called when the nested transaction aborts, but the call need only be made once for a transaction family when that family commits (recall that nested transaction commits are handled implicitly by the Concurrency Control Service).

4.5.3 The LockSet Interface

For clients operating in the implicit mode, locks are acquired and released on lock sets which are defined by means of the **LockSet** interface. The **LockSet** interface only provides operations to acquire and release locks on behalf of the calling thread or transaction. The interface does not provide support for transactional clients that use the explicit Transaction Service interfaces.

```
interface LockSet {
    void lock(in lock_mode mode);

    boolean try_lock(in lock_mode mode);

    void unlock(in lock_mode mode)
        raises(LockNotHeld);

    void change_mode(in lock_mode held_mode,
                    in lock_mode new_mode)
        raises(LockNotHeld);

    LockCoordinator get_coordinator(in
        CosTransactions::Coordinator which);
};
```

When calls to acquire or release locks are made outside the scope of a transaction then it is assumed that the client is operating in the non-transactional mode (the Concurrency Control Service implementation must use the appropriate Transaction Service operation to determine whether the current thread is executing on behalf of a transaction).

- **lock()**

This acquires a lock on the specified lock set in the specified mode. If a lock is held on the same lock set in an incompatible mode by another client, then the operation will block the calling thread of control until the lock is acquired. If a call that is on behalf of a transactional client is blocked and the transaction is aborted, then the call will return with the **Transactions::TransactionRolledBack** exception.

- **try_lock()**

This attempts to acquire a lock on the specified lock set. If the lock is already held in an incompatible mode by another client, then the operation returns a **FALSE** result to indicate that the lock could not be acquired.

- **unlock()**

This drops a single lock on the specified lock set in the specified mode (recall that a lock can be held multiple times in the same mode). Calls to drop a lock that is not held result in the **LockNotHeld** exception being raised

- **change_mode()**

This changes the mode of a single lock (recall that multiple locks may be held on the same lock set). If the new mode conflicts with an existing mode held by an unrelated client, then the **change_mode()** operation blocks the calling thread of control until the new mode can be granted. Like the lock call, if the client is a transaction and it aborts while the thread of control is blocked, then the **Transactions::TransactionRolledBack** exception will be raised. Similarly, when a call is made to change the mode of a lock, but the lock is not held in the specified mode, the **LockNotHeld** exception will be raised.

- **get_coordinator()**

This returns the **LockCoordinator** associated with the specified transaction.

4.5.4 The TransactionalLockSet Interface

The **TransactionalLockSet** interface provides operations to acquire and release locks on a lock set on behalf of a specific transaction. The operations that make up the **TransactionalLockSet** interface are:

```
interface TransactionalLockSet {
    void lock(in CosTransactions::Coordinator which,
             in lock_mode mode);

    boolean try_lock(in CosTransactions::Coordinator which,
                    in lock_mode mode);

    void unlock(in CosTransactions::Coordinator which,
               in lock_mode mode)
        raises(LockNotHeld);

    void change_mode(in CosTransactions::Coordinator which,
                    in lock_mode held_mode,
                    in lock_mode new_mode)
        raises(LockNotHeld);

    LockCoordinator get_coordinator(in
        CosTransactions::Coordinator which);
};
```

The operations provided by the **TransactionalLockSet** interface operate in an identical manner to the equivalent operations provided by the **LockSet** interface. The interfaces differ in that for the **TransactionalLockSet** interface the identity of the transaction is passed explicitly as a reference to the coordinator for the transaction instead of implicitly through an association with the calling thread.

4.5.5 The LockSetFactory Interface

Lock sets are created using the **LockSetFactory** interface.

```
interface LockSetFactory {
    LockSet create();
    LockSet create_related(in LockSet which);

    TransactionalLockSet create_transactional( );
    TransactionalLockSet
        create_transactional_related(in
            TransactionalLockSet which);
};
```

This interface provides two sets of operations that return new **LockSet** and **TransactionalLockSet** instances.

- **create()**
This creates a new lock set and lock **Coordinator**.
- **create_related()**
This creates a new lock set that is related to an existing lock set. Related lock sets drop their locks together.
- **create_transactional()**
This creates a new transactional lock set and lock **Coordinator** for explicit mode transactional clients.
- **create_transactional_related()**
This creates a new transactional lock set that is related to an existing lock set. Related lock sets drop their locks together.

Externalization Service Specification

5.1 Service Description

The Externalization Service specification defines protocols and conventions for *externalizing* and *internalizing* objects. To externalize an object is to record the object's state in a stream of data. Objects which support the appropriate interfaces and whose implementations adhere to the proper conventions can be externalized to a **Stream** (in memory, on a disk file, across the network, and so on) and subsequently be internalized into a new object in the same or a different process. The externalized form of the object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and can be internalized in a different, disconnected ORB.

Many different externalized data formats and storage mediums can be supported by service implementations. But, for portability, clients can request that externalized data be stored in a file using a standardised format that is defined as part of this Externalization Service specification.

Externalizing and internalizing an object are similar to copying the object. The **copy()** operation creates a new object that is initialized from an existing object. The new object is then available to provide service. Furthermore, with the **copy()** operation, there is an assumption that it is possible to communicate via the ORB between the *here* and *there*. Externalization, on the other hand, does not create an object that is initialized from an existing object. Externalization "stops along the way". New objects are not created until the **Stream** is internalized. Furthermore, there is no assumption that it is possible to communicate via the ORB between *here* and *there*.

The Externalization Service is related to the Relationship Service. It also parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations (for more information, see Section 2.5 on page 15.)

The Externalization Service defines Protocols in these areas:

- Client's view of externalization, composed of the interfaces used by a client to externalize and internalize objects. The client's view of externalization is defined by the **Stream** interface.
- Object's view of externalization, composed of the interfaces used by an externalizable object to record and retrieve their object state to and from the **Stream's** external form. The object's view is defined by the **StreamIO** interface.
- **Stream's** view of externalization, composed of the interfaces used by the **Stream** to direct an externalizable object or graph of objects to record or retrieve their state from the **Stream's** external form. The **Stream's** view of externalization is given by the **Streamable**, **Node**, **Role** and **Relationship** interfaces.

5.2 Service Structure

This section explains the model of externalization for client and **Stream**. It also describes the model of externalization and internalization for objects.

5.2.1 Client Model of Externalization

A client has a simple view of the Externalization Service. A client that wishes to externalize an object first must have an object reference for a **Stream** object. A **Stream** object owns and provides access to the externalized form of one or more objects. **Streams** may be provided that hold externalized data on various mediums such as in memory or on disk. All Externalization Service implementors provide a **Stream** object that saves the externalized data in a file. A client may create a **Stream** object using the **create()** operation on a **StreamFactory** object, or may specify that a file be used to store the externalized data using the **create()** operation of a **FileStreamFactory** object.

The client can create a **Stream** object that supports a standardised externalization data format. Externalization data that follows this format will be internalizable on all CORBA-compliant ORBs that can locate compatible object implementations. By including support for a specific external representation format in the Externalization Service, portability of object state is provided across different CORBA-compliant implementations and hardware architectures.

Once a client has a **Stream** object, the client may externalize an object by issuing an **externalize()** request on the **Stream** object, providing the object reference to the object that should be externalized. In general, the client is unaware of whether externalizing an object causes any other related objects to be externalized. An externalizable object may represent a simple object, a set of objects, or a graph of related objects. The client uses the same interface in all cases.

If a client wishes to externalize multiple objects (or related sets of objects) to the same **Stream**, the client issues a **begin_context()** request before the first externalize request, and then issues an **end_context()** following the last externalize request for that same **Stream**.

The externalized form of the object can exist in the **Stream** object for arbitrary amounts of time, be transported by means outside of the ORB, and can be internalized in a different, disconnected ORB.

A client that wishes to internalize an object issues an **internalize()** request on the appropriate **Stream** object, providing a factory finder. The **Stream** object interacts with the specified factory finder, or uses other implementation-dependent mechanisms, to create an implementation of the object that matches the externalized data. The client is returned an object reference to the newly internalized object.

5.2.2 Stream Model of Externalization

A **Stream** object provides the **Stream** interface for use by clients. The **Stream** object is also responsible for providing an object that supports a **StreamIO** interface for actually reading and writing data to the externalized data form. The **Stream** object may support the **StreamIO** interfaces itself, or may create another object that supports the **StreamIO** interfaces. This is considered an implementation detail.

Note: When the behaviour described in this section may be implemented in either the **Stream** or **StreamIO** objects (or other internal objects they may use), the term **Stream service** is used.

When a **Stream** object receives an externalize request from a client, it also gets an object reference to the object to be externalized. The **Stream** cooperates with the externalizable object to accomplish externalization and internalization, using the object's **Streamable** interfaces.

The **Stream** service uses the read-only **Key** attribute of the externalizable object to decide what information to put into the external data in order to be able to find the correct factory and implementation with which to subsequently internalize an equivalent object. The **Stream** service then issues an **externalize_to_stream()** request to the externalizable object, providing an object reference to a **StreamIO** object that is to be used by the externalizable object to record its state in the **Stream** service's external data.

When a **Stream** object receives an internalize request from a client, it also gets a factory finder. The **Stream** service holds the external form of the object, or set of objects, to be internalized. The **Stream** service reads the key from its externalized data. It may then pass the key to the factory finder to locate a factory that can create an object with an implementation that matches the recorded object state. The **Stream** service implementation may use other implementation-specific ways of creating an appropriate object. The **Stream** service then issues an **internalize_from_stream()** request to the newly created object, providing an object reference to a **StreamIO** object that is used by the externalizable object to initialize its state according to the **Stream** service's externalized data.

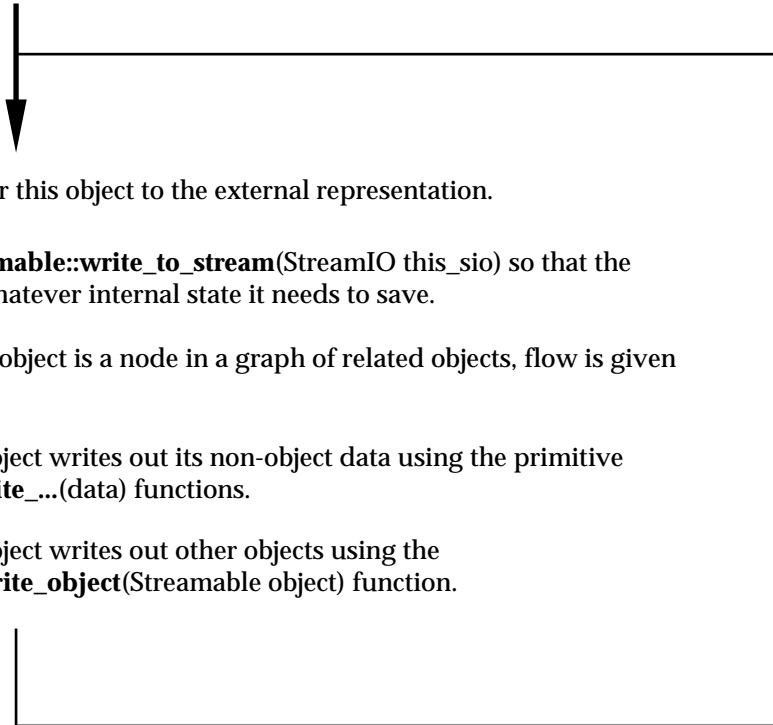
When a **Stream** object receives a **begin_context()** request, the **Stream** service sets up a context during which the **Stream** service ensures that externalizing multiple objects that may have overlapping object references and/or object relationships produces single instances of those objects on internalization. An **end_context()** request causes the **Stream** service to remove the previous internal context, and externalize subsequent objects without regard to whether they have already been externalized in this **Stream**'s data.

5.2.3 Object Model of Externalization

Every object that wishes to be externalizable must support the **Streamable** interface, and follow conventions on use of the **StreamIO** interfaces to record and retrieve their object state from a **Stream**'s data.

When a **Streamable** object receives an **externalize_to_stream()** request from the **Stream** service, it must write all of its state necessary for internalization to the **StreamIO** object provided by the **Stream** service. **StreamIO** provides **write_<type>()** operations for writing each of the CORBA basic data types, plus string types. If an object has object references that are part of its state, the **StreamIO write_object()** operation may be used to cause the object specified by an object reference to also be externalized to the **Stream**'s data.

Client calls **Stream::externalize** (Streamable object)



Stream writes a key for this object to the external representation.

Stream calls the **Streamable::write_to_stream**(StreamIO this_sio) so that the object can write out whatever internal state it needs to save.

If **Streamable** object is a node in a graph of related objects, flow is given in Figure 5-2.

Streamable object writes out its non-object data using the primitive **StreamIO::write...**(data) functions.

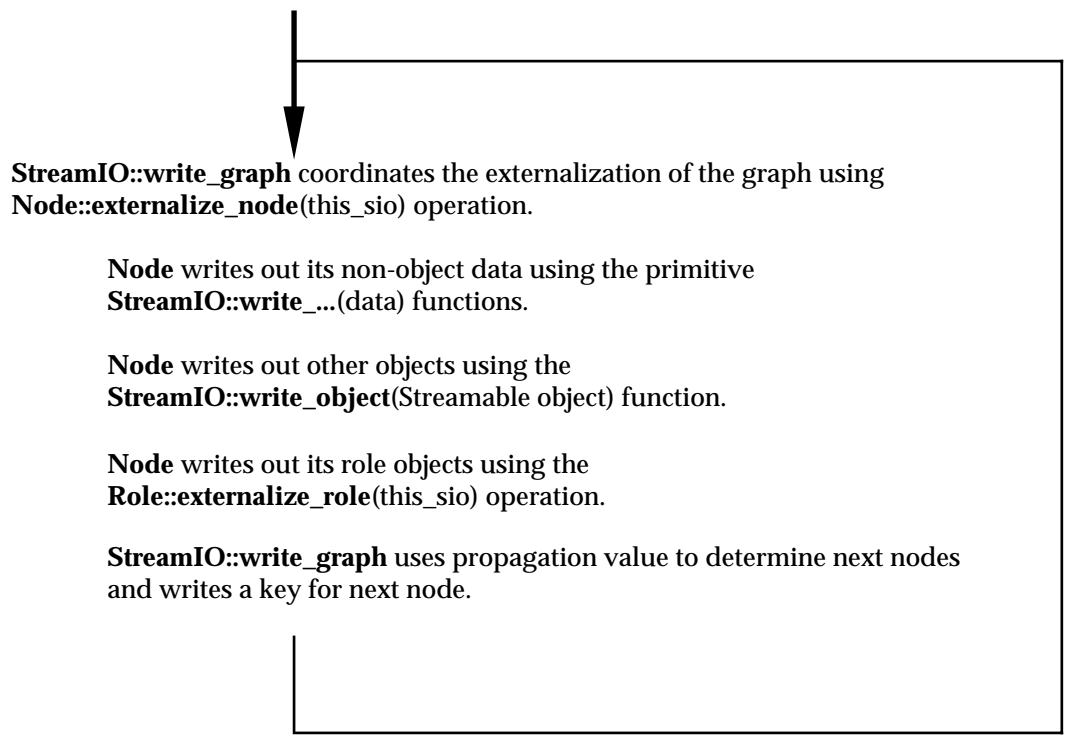
Streamable object writes out other objects using the **Stream-IO::write_object**(Streamable object) function.

Figure 5-1 Externalization Control Flow (Streamable Object is Not a Node)

A **Streamable** object may be a node in a graph of related objects; that is, it may use the Relationship Service to connect to other objects and support the **CosCompoundExternalization::Node** interface. Such a **Streamable** object simply delegates the **Streamable::externalize_to_stream()** request back to the **Stream** service, using the **StreamIO::write_graph()** operation.

The **Stream** service then coordinates the externalization of the graph and calls the object back using the object's **CosCompoundExternalization::Node** interface.

Streamable object, recognising that it is a node in a graph of related objects, delegates the externalization of the graph to the stream service using **StreamIO::write_graph(this_node)** operation.



StreamIO::write_graph coordinates the externalization of the graph using **Node::externalize_node(this_sio)** operation.

Node writes out its non-object data using the primitive **StreamIO::write_...(data)** functions.

Node writes out other objects using the **StreamIO::write_object(Streamable object)** function.

Node writes out its role objects using the **Role::externalize_role(this_sio)** operation.

StreamIO::write_graph uses propagation value to determine next nodes and writes a key for next node.

StreamIO object externalizes the involved relationships using **Relationship::externalize()**. **StreamIO** writes traversal scoped IDs for the externalized roles and relationships to the Stream's data.

Figure 5-2 Externalization Control Flow (Streamable Object is a Node)

5.2.4 Object Model of Internalization

When a **Streamable** object receives an **internalize_from_stream()** request from a **Stream**, it must read data from the **StreamIO** object provided by the **Stream** service, and initialize its state to match the externalized state. The externalizable object requests data from the **Stream** service using the **StreamIO read_<type>()** operation for basic data and string types. If the object being internalized includes a reference to another object as part of its state, the **StreamIO read_object()** operation may be used to have that object also internalized from the **Stream's** data.

Client calls **Streamable = Stream::internalize(FactoryFinder f)**



Stream reads key from the external representation, and uses this and the factory finder to create an object of the correct interface and implementation. The stream may use the **StreamableFactory** interface.

Stream calls the **Streamable::read_from_stream(StreamIO this_sio)** so that the object can read the data in its external representation and reset or calculate its internal state.

If **Streamable** object is a node in a graph of related objects, flow is given in Figure 5-4.

Streamable object reads in its non-object data using the primitive **StreamIO::read_...(data)** functions.

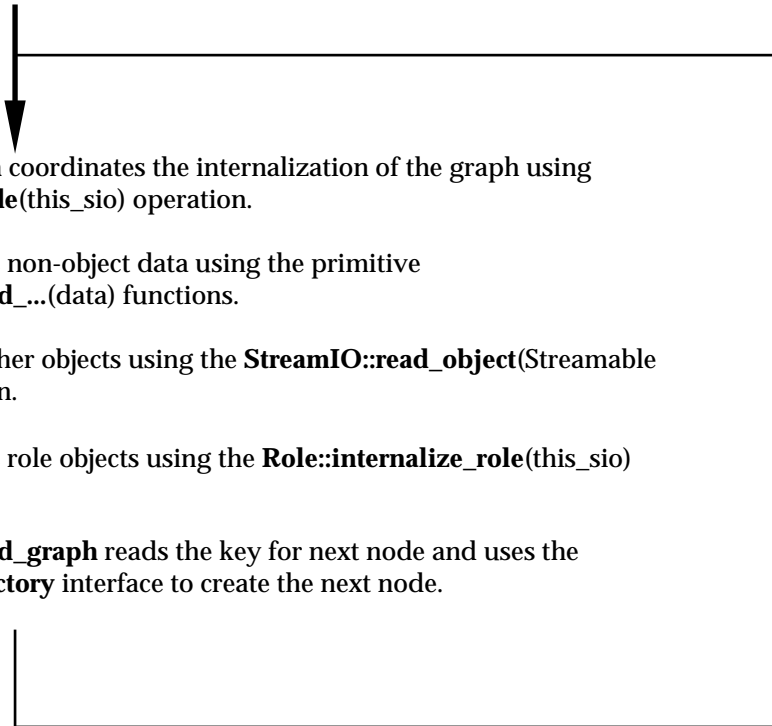
Streamable object internalizes other objects using the **Streamable = StreamIO::read_object()** function.

Figure 5-3 Internalization Control Flow (Streamable Object is Not a Node)

A **Streamable** object may be a node in a graph of related objects; that is, it may use the Relationship Service to connect to other objects and support the **CosCompoundExternalization::Node** interface. Such a **Streamable** object simply delegates the **Streamable::internalize_from_stream()** request back to the **Stream** service, using the **StreamIO::write_graph()** operation.

The **Stream** service then coordinates the externalization of the graph and calls the object back using the object's **CosCompoundExternalization::Node** interface.

Streamable object, recognising that it is a node in a graph of related objects, delegates the internalization of the graph to the stream service using **StreamIO::read_graph(this_node)** operation.



StreamIO object internalizes the traversal scoped identifiers for the externalized roles and relationships and internalizes the relationships using **Relationship::internalize()**.

Figure 5-4 Internalization Control Flow (Streamable Object is a Node)

5.3 Object and Interface Hierarchies

This section identifies the objects required for the Externalization Service and important inheritance and use relationships that exist between their interfaces.

The Externalization Service can only externalize and internalize objects that inherit the **Streamable** interface. **Streamable** does not inherit any other interfaces. However, it must have an associated **StreamableFactory** that the Externalization Service implementation can find and use when internalizing the object.

Stream inherits the **LifeCycleObject** interface because clients of the Externalization Service need to remove these objects. The **StreamFactory** or **FileStreamFactory** interfaces may be used to create **Stream** objects.

In addition to the inheritance relationships described above, the class diagram in Figure 5-5 on page 83 also shows the usage relationships between the service objects. **Stream::externalize()** and **internalize()** operations invoke the **Streamable externalize_to_stream()** and **internalize_from_stream()** operations to write and read the appropriate object internal state. A **StreamIO** object is passed as an argument to these operations. The externalized object determines how much of its state must be put in the external representation, and can minimise saved state by recreating some state upon internalization. The **Streamable externalize_to_stream()** and **internalize_from_stream()** use **StreamIO** operations to actually put various data types and contained object references in the external representation. This allows **StreamIO** to put appropriate headers in the external representation so that the object can be recreated correctly during internalization. The **Stream** is responsible for providing an object that supports the **StreamIO** interface. The **Stream** object may support the **StreamIO** interface itself, or create another object that supports the **StreamIO** interface. The **Stream** and **StreamIO** implementations decide on the storage medium and data type representation conversion for different hardware, without requiring different implementation of the objects being externalized.

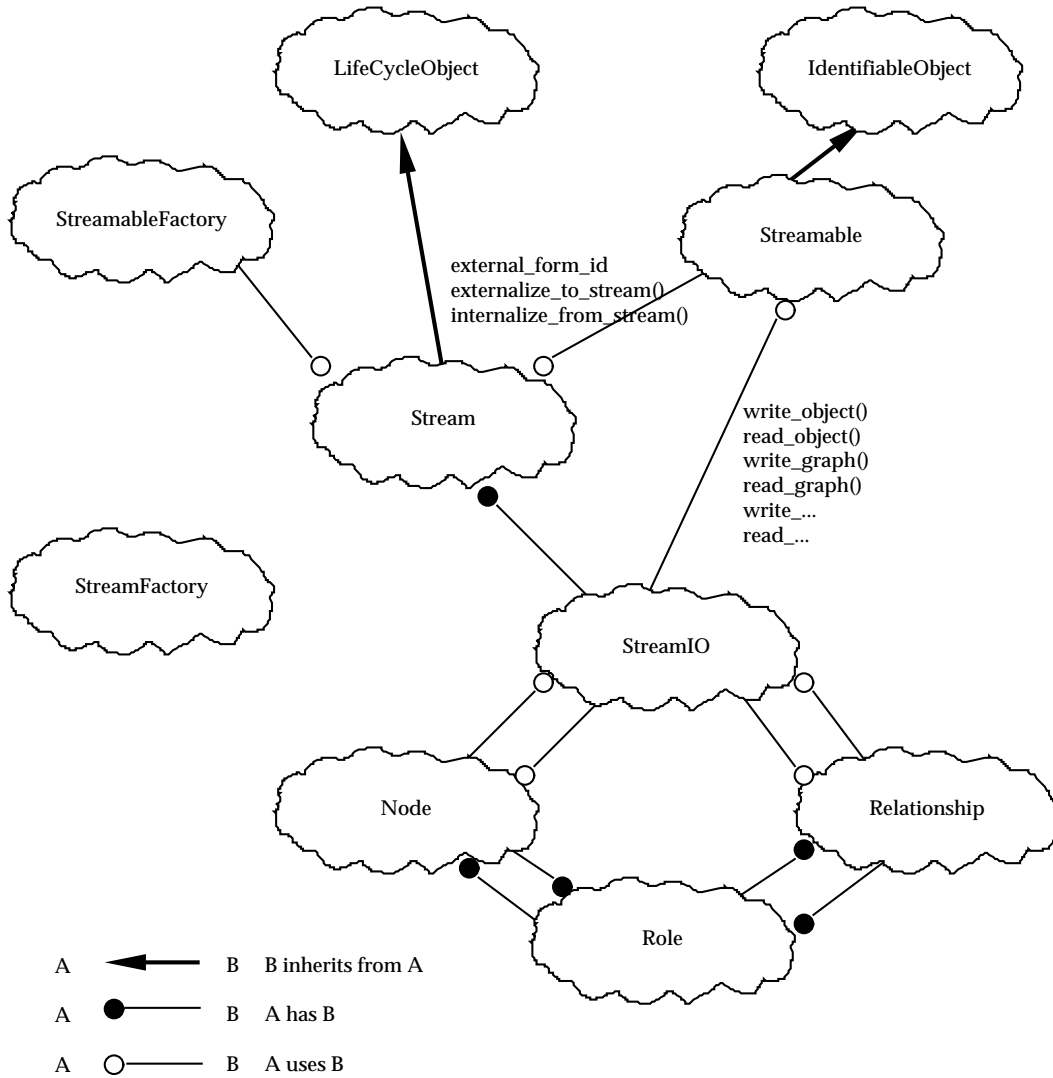


Figure 5-5 Object Externalization Service Booch Class (=Interface)

5.4 Interface Summary

The Externalization Service defines interfaces (using OMG IDL) to support the functionality described in the previous sections. The following tables give high-level descriptions of the Externalization Service interfaces. Subsequent sections describe the interfaces in more detail.

Interface	Purpose	Primary Client
Stream	Holds external form of objects.	Clients that need to externalize and internalize objects.
StreamFactory	Creates and initializes Stream objects.	Clients that need to create Stream objects.
FileStreamFactory	Creates and initializes Stream objects that store data in a file.	Clients that need to create Stream objects, and want the externalized data in a file.

Table 5-1 Client Functional Interfaces

Table 5-1 shows the client functional interfaces which support the client model of externalization.

Interface	Purpose	Primary Client
Streamable	Provides its state to a Stream for externalization, and gets its state from the Stream on internalization.	The Stream service implementation of externalization and internalization.
StreamableFactory	Creates and initializes Streamable objects.	The Stream service internalization implementation.
StreamIO	Part of Stream implementation that writes and reads object state to apparently converted external form.	The externalizable objects that need to record and retrieve their state from a Stream .

Table 5-2 Service Construction Interfaces

Table 5-2 lists the service construction interfaces which support the service implementation model of externalization.

Interface	Purpose	Primary Client
Node	Defines externalization and internalization operations on nodes on graphs of related objects.	The Stream service implementation of externalization and internalization.
Relationship	Defines externalization and internalization operations on relationships.	The Stream service implementation of externalization and internalization.
Role	Defines externalization and internalization operations on relationships.	The Stream service implementation of externalization and internalization.

Table 5-3 Compound Externalization Interfaces

Table 5-3 lists the compound externalization interfaces which support the service implementation model graph of externalization.

Externalization Service Architecture: Audience/Bearer Mapping

Stream and **StreamFactory** are solely functional interfaces. Their audience is the client of the Externalization Service.

Streamable, **StreamableFactory** and **StreamIO** are solely construction interfaces. The audience for **Streamable** is both the **Stream** and **StreamIO** objects. To be externalizable, objects must inherit the **Streamable** interface and provide implementations of its operations. The audience for the **StreamIO** interface is the externalizable **Streamable** and **StreamableNode** objects. The **StreamIO** objects are part of the Externalization Service implementation.

The **Stream**, **StreamFactory** and **StreamIO** objects are specific objects because their purpose is to provide a part of the Externalization Service. However, there may be many **Stream** and **StreamIO** instances in a system, since each represents a particular external representation of an object or group of objects.

Streamable and **StreamableFactory** objects are generic objects because their primary purpose is unrelated to the Externalization Service. Any definer or implementor of an object may choose to inherit the **Streamable** interface in order to support externalization/internalization of that object.

In summary:

- **Stream** and **StreamFactory** are specific functional interfaces.
- **Streamable** and **StreamableFactory** are generic construction interfaces.
- **StreamIO** is a specific construction interface.

5.5 The CosExternalization Module

The client-functional interfaces defined by the **CosExternalization** module are:

- **StreamFactory** interface, which creates a **Stream**.
- **FileStreamFactory** interface, which has an operation that lets clients cause externalized data be stored in a file or internalize objects from a file they have been given.
- **Stream** interface, which can externalize one object or a group of objects, finalise the externalization, and internalize an object.

```
#include <LifeCycle.idl>
#include <Stream.idl> module CosExternalization {
    exception InvalidFileNameError{};
    exception ContextAlreadyRegistered{};
    interface Stream: CosLifeCycle::LifeCycleObject{
        void externalize(
            in CosStream::Streamable theObject);
        CosStream::Streamable internalize(
            in CosLifeCycle::FactoryFinder there)
            raises( CosLifeCycle::NoFactory,
                CosStream::StreamDataFormatError );
        void begin_context( )
            raises( ContextAlreadyRegistered);
        void end_context();
        void flush();
    };
    interface StreamFactory {
        Stream create();
    };
    interface FileStreamFactory {
        Stream create(
            in string theFileName)
            raises( InvalidFileNameError );
    };
};
```

5.5.1 The StreamFactory Interface

Creating a Stream Object

```
stream create();
```

Clients of the Externalization Service must create a **Stream** object before they can externalize or internalize any objects. Two factory interfaces are supported: the **StreamFactory** interface and the **FileStreamFactory** interface. The **StreamFactory** interface has a **create()** operation that creates a **Stream** without specifying any special characteristics of the implementation.

5.5.2 The FileStreamFactory Interface

Creating a Stream Object

```
Stream create(
    in string theFileName)
    raises( InvalidFileNameError );
```

For clients that want to cause the externalized data stored in a file, or that need to internalize objects from a file they have been given, the **FileStreamFactory** interface has a **create()** operation that takes a string input parameter. The client sets this string to the filename of the file that will be used by the **Stream** service to hold the external representation of the objects externalized, or that contains the external representation of objects that the client wishes to internalize. **Stream::externalize()** requests will append to any existing data in the file associated with a **Stream**.

5.5.3 The Stream Interface

Externalizing an Object

```
void externalize(in CosStream::Streamable theObject);
```

Clients of the Externalization Service invoke **externalize()** on a **Stream** object passing the object reference of a **CosStream::Streamable** object, **theObject**, to be externalized. Only objects that are of type **CosStream::Streamable** can be externalized. Subsequently, clients invoke the **internalize()** operation on the **Stream** containing the external representation, and **Stream internalize()** operation creates a new object with state identical to what was externalized and returns the new object reference.

The implementation of **externalize()** writes implementation-specific header information to the external representation it is maintaining, so that the correct object can be recreated at internalization time. This could be the factory key that was used to create the **CosStream::Streamable** object, or could include the interface type, implementation repository, or factory object names. The factory key may be obtained by from the **external_form_id** attribute of **theObject**. The **externalize()** implementation must then invoke the **CosStream::Streamable externalize_to_stream()** operation on **theObject** to cause the object's internal state to be written to the external representation. The **Stream** is responsible for providing an object that supports the **StreamIO** interfaces for the externalizable object to use in writing data to the **Stream** service.

Externalizing Groups of Objects

```
void begin_context()
    raises( ContextAlreadyRegistered );
void end_context();
```

If a client wishes to externalize a set of objects with overlapping references and/or object relationships, the client invokes **begin_context()** on the **Stream**. This must be called before externalizing any of the set of objects, and **end_context()** must be called on the **Stream** after the entire set of objects has been externalized and before the **Stream** is used with another set of objects.

The **Stream** implementation establishes an association with the specified **Stream** object and a logical context. The **Stream** ensures that all objects externalized to this **Stream** while this association lasts will be externalized in such a way that internalization will not create any duplicate objects. That is, the implementation of **Stream** checks for context, and for objects externalized in the same context handles overlapping or circular references and/or relationships

between those objects. The association lasts until `end_context()` is called. The **Stream** raises the **ContextAlreadyRegistered** exception if `begin_context()` is called and a context is already established, perhaps through some other implementation-dependent mechanism or perhaps because `end_context()` has not been called following a previous `begin_context()`.

Completing Externalization

```
void flush ();
```

Clients invoke `flush()` to request that the external representation is committed to its final storage medium, whatever that may be. The implementation of `flush()` should attempt to ensure that the external representation is completely up-to-date in its final storage (for example, memory buffer, file, tape, and so on).

Internalizing an Object

```
CosStream::Streamable internalize(
    in CosLifeCycle::FactoryFinder there)
    raises(CosLifeCycle::NoFactory,
          CosStream::StreamDataFormatError );
```

The implementation of `internalize()` must create an object with the correct interface and implementation to match the externalized representation and return a pointer to the new **CosStream::Streamable** object. The `internalize()` implementation must then invoke the `internalize_from_stream()` operation on the new object. The **CosStream::StreamDataFormatError** exception should be raised if an error is detected in the data format of the object header. The **CosLifeCycle::NoFactory** exception should be raised if the object cannot be created because an appropriate factory cannot be found. If the object cannot be created due to other reasons, an **ObjectCreationError** exception should be raised. Additional **CosStream::StreamDataFormat** exceptions may be raised by the `read_<type>()` operations invoked by `internalize_from_stream()` operation due to errors in the externalized data format.

5.6 The CosStream Module

The service construction interfaces defined by the **CosStream** module are:

- **Streamable** interface
- **StreamableFactory** interface
- **StreamIO** interface.

```
#include <LifeCycle.idl>
#include <ObjectIdentity.idl>
#include <CompoundExternalization.idl>
module CosStream {
    exception ObjectCreationError{};
    exception StreamDataFormatError{};
    interface StreamIO;

    interface Streamable:
        CosObjectIdentity::IdentifiableObject {
            readonly attribute CosLifeCycle::Key external_form_id;
            void externalize_to_stream(
                in StreamIOtargetStreamIO);
            void internalize_from_stream(
                in StreamIOSourceStreamIO,
                in FactoryFinder there);
            raises( CosLifeCycle::NoFactory,
                ObjectCreationError,
                StreamDataFormatError );
        };

    interface StreamableFactory {
        Streamable create_uninitialized( );
    };

    interface StreamIO {
        void write_string(in string aString);
        void write_char(in char aChar);
        void write_octet(in octet anOctet);
        void write_unsigned_long(
            in unsigned long anUnsignedLong);
        void write_unsigned_short(
            in unsigned short anUnsignedShort);
        void write_long(in long aLong);
        void write_short(in short aShort);
        void write_float(in float aFloat);
        void write_double(in double aDouble);
        void write_boolean(in boolean aBoolean);
        void write_object(in Streamable aStreamable);
        void write_graph(in CosCompoundExternalization::Node);
        string read_string( )
            raises(StreamDataFormatError);
        char read_char( )
            raises(StreamDataFormatError );
        octet read_octet( )
            raises(StreamDataFormatError );
        unsigned long read_unsigned_long( )
            raises(StreamDataFormatError );
        unsigned short read_unsigned_short( )
            raises( StreamDataFormatError );
    };
};
```

```

    long read_long()
        raises(StreamDataFormatError );
    short read_short()
        raises(StreamDataFormatError );
    float read_float()
        raises(StreamDataFormatError );
    double read_double()
        raises(StreamDataFormatError );
    boolean read_boolean()
        raises(StreamDataFormatError );
    Streamable read_object(
        in FactoryFinder there,
        in Streamable aStreamable)
        raises(StreamDataFormatError );
    void read_graph(
        in CosCompoundExternalization::Node starting_node,
        in FactoryFinder there) raises(StreamDataFormatError );
};

```

5.6.1 The StreamIO Interface

The `write_<type>()` and `read_<type>()` operations on **StreamIO** are used by **Streamable** `externalize_to_stream()` and `internalize_from_stream()` operations to cause internal object state to be written to or read from the external representation. The `externalize_to_stream()` decomposes the internal state of an object in a series of primitive data type values that can be written and read with these operations. **StreamIO** supports writing and reading all the CORBA basic data types.

The implementation of the `write...()` and `read...()` operations are responsible for any desired conversion of the data and transferring the data to or from the desired external representation. Actual transfer of the representation to the final storage medium may be deferred until the `flush()` operation. All details of the external representation format, storage medium and buffering are specific to the implementation. Different implementations may support buffering of the external representation data in memory, converting data values to a canonical binary form for exchange across big/little endian CPU hardware, conversion of data to a canonical text form for readability, or to facilitate mailing objects across networks, use of various storage mediums such as memory, filesystem, tape or other differences. See Section 5.11 on page 100 for information on a portable external representation. A **StreamDataFormatError** exception should be raised if errors are detected in the data format of the external representation.

In support of integrating the Externalization Service with the Transaction and Persistent Object Services, the `read_object()` operation supports the internalization to existing objects. The semantics of the operation are that if the **Streamable** parameter is **Null**, then the **FactoryFinder** parameter is used to create an instance for internalize. If the **Streamable** parameter is not **Null**, then the **StreamIO** implementation will internalize to the **Streamable** object. This semantic allows the Externalization Service to be used as a Persistent Object Service protocol and to support the `restore()` operation on existing objects in the case of an aborted transaction.

5.6.2 The Streamable Interface

Object implementors must inherit from the **Streamable** interface if they want an object to be externalizable. Three operations must be implemented.

Comparing Streamable Objects

```
boolean CosObjectIdentity::IdentifiableObject::is_identical(
    in CosObjectIdentity::IdentifiableObject anObject);
```

```
readonly unsigned long constant_random_id;
```

A **Streamable** object inherits from **CosObjectIdentity::IdentifiableObject**, and therefore must support a **constant_random_id** attribute and an **is_identical()** operation. The **Stream** service uses these to compare objects when detecting cycles or overlapping references in objects being externalized to the same **Stream** in the same context or within the same graph. The **constant_random_id** attribute value does not have to be unique, but a unique value may substantially speed up the externalization process.

Creation Key for a Streamable Object

```
readonly attribute CosLifeCycle::Key external_form_id;
```

A **Streamable** object must support a read-only attribute, **external_form_id**, which is a key that can be given to a factory finder in order to find a factory that could have created this object. The **Stream** service may use this attribute during internalization to create an object that can reinitialize itself from the externalized data.

Writing the Object State to a Stream

```
void externalize_to_stream(
    in StreamIOtargetStreamIO);
```

The **externalize_to_stream()** operation is responsible for decomposing an externalizable object's internal state into a series of primitive data type values and object references. The **externalize_to_stream()** function must write out all the necessary primitive data values using the **write_<type>()** operations on the **targetStreamIO** for non-object data types. If this object has other object references, then, normally, those objects should also be written out using the **write_object()** operation on the **targetStreamIO**. However, it is up to the **Streamable** implementor to decide which referenced objects should be externalized with this object. The primitive data values must all be written before any of the embedded object references are written.

If the **Streamable** object is a node in a graph, then it should delegate the **externalize_to_stream()** to the **StreamIO** by invoking **write_graph()**. The object would subsequently receive an **externalize_node_to_stream()** and write out its internal state as described above. **Node** objects should not call **write_object()** for other nodes in their graph, but may call **write_object()** for object references that are not for nodes in their graph.

Reinitializing the Object State from a Stream

```
void internalize_from_stream(
    in StreamIOsourceStreamIO,
    in FactoryFinder there);
```

The **internalize_from_stream()** operation is responsible for reinitializing the object's internal state from the series of primitive data type values and object references that are written/flattened during **externalize_to_stream()**. The **internalize_from_stream()** operation should read in all the necessary internal state of the object using the **read_<type>()** operations on the **sourceStreamIO**

for non-object data types. If this object has other object references that were externalized using `write_object()`, then those objects should be recreated using the `read_object()` operation on the `sourceStreamIO` with the same `FactoryFinder` argument as the `there` parameter passed in to the `internalize_from_stream()` operation. The `read_<type>()` and `read_object()` operations for the various portions of the object's internal state must be invoked in the same order in which they are written by the `externalize_to_stream()` implementation. The `internalize_from_stream()` must also initialize any additional state that was not externalized because it can be derived from other state information. Therefore, the `externalize_to_stream()` and `internalize_from_stream()` operations must be designed to complement each other.

If the `Streamable` object is a node in a graph, then it should delegate the `internalize_to_stream()` to the `sourceStreamIO` by invoking `read_graph()` with the same `FactoryFinder` argument as the `there` parameter passed in to the `internalize_from_stream()` operation. The `Streamable` (also `Node`) object would subsequently receive an `internalize_node_to_stream()` and read in its internal state as described above. `Node` objects should not call `read_object()` for other nodes in their graph, but may call `read_object()` for object references that are not for nodes in their graph.

The `ObjectCreationError` and `StreamDataFormatError` exceptions originate from the `read_<type>()` operations on the `sourceStreamIO`, and are not explicitly raised by the `internalize_from_stream()` code.

5.6.3 The StreamableFactory Interface

Creating a Streamable Object

`Streamable create_uninitialized();`

The `Stream` service must be able to create a `Streamable` object in order to internalize an object from the `Stream`'s externalized data. For any externalizable object, a `StreamableFactory` object must exist that supports creation of that object. This factory must be findable using the read-only `external_form_id Key` attribute of the `Streamable` object. The `Stream` service implementation could store this key during externalization and use it during internalization to find the factory that can create the externalized object. However, a `Stream` implementation may use other means to create the object during internalization. The `create_uninitialized()` operation on the `StreamableFactory` should create the associated `Streamable` object. This `Streamable` object does not have to be initialized, since that can be done on the subsequent `internalize_from_stream()` operation on the newly created `Streamable` object.

5.7 The CosCompoundExternalization Module

If a **Streamable** object participates as a node in a graph of related objects, the **Streamable** object can delegate the externalization operation to the **Stream** service. In particular, the **Streamable** object simply uses the **write_graph()** operation. The **write_graph()** operation expects a **Streamable** object reference as a starting node. The **Stream** service narrows the **Streamable** object reference to **CosCompoundExternalization::Node**. The **write_graph()** then coordinates the orderly externalization of the graph of related objects. For more details on compound operations, see Chapter 6 on page 103 and the Life Cycle Service specification in Common Object Services, Volume 1.

The **CosCompoundExternalization** module defines the **Node**, **Role**, **Relationship** and **PropagationCriteriaFactory** interfaces for use by the **write_graph()** operation.

The **CosCompoundExternalization** module is shown below. Detailed descriptions of the interfaces follow.

```
#include <Graphs.idl>
#include <Stream.idl>

module CosCompoundExternalization {
    interface Node;
    interface Role;
    interface Relationship;
    interface PropagationCriteriaFactory;

    struct RelationshipHandle {
        Relationship theRelationship;
        ::CosObjectIdentity::ObjectIdentifier constantRandomId;
    };
    interface Node : ::CosGraphs::Node, ::CosStream::Streamable{
        void externalize_node (in ::CosStream::StreamIO sio);
        void internalize_node (in ::CosStream::StreamIO sio,
            in ::CosLifeCycle::FactoryFinder there,
            out Roles rolesOfNode)
            raises (::CosLifeCycle::NoFactory);
    };

    interface Role : ::CosGraphs::Role {
        void externalize_role (in ::CosStream::StreamIO sio);
        void internalize_role (in ::CosStream::StreamIO sio);
        ::CosGraphs::PropagationValue externalize_propagation (
            in RelationshipHandle rel,
            in ::CosRelationships::RoleName toRoleName,
            out boolean sameForAll);
    };
    interface Relationship :
        ::CosRelationships::Relationship {
        void externalize_relationship (
            in ::CosStream::StreamIO sio);
        void internalize_relationship(
            in ::CosStream::StreamIO sio,
            in ::CosGraphs::NamedRoles newRoles);
        ::CosGraphs::PropagationValue externalize_propagation (
            in ::CosRelationships::RoleName fromRoleName,
            in ::CosRelationships::RoleName toRoleName,
            out boolean sameForAll);
    };
};
```

```

interface PropagationCriteriaFactory {
    ::CosGraphs::TraversalCriteria create_for_externalize( );
};

```

5.7.1 The Node Interface

The **Node** interface defines operations to internalize and externalize a node.

Externalizing a Node

```
void externalize_node (in ::CosStream::StreamIO sio);
```

The `externalize_node()` operation transfers the **Node**'s state to the **Stream** given by the `sio` parameter. It is the **Node**'s responsibility to externalize its roles as well. The **Node** can accomplish this by writing the role's key to the **Stream** and using the `Role::externalize_role()` operation.

Internalizing a Node

```
void internalize_node (in ::CosStream::StreamIO sio,
                    in ::CosLifecycle::FactoryFinder there,
                    out Roles rolesOfNode)
    raises (::CosLifecycle::NoFactory);
```

The `internalize_node()` operation causes a **Node** and its roles to be internalized from the **Stream** `sio`.

It is the **Node**'s responsibility to create and internalize its roles. It can do this by reading the key for a role from the **Stream** and using the `CosStream::StreamableFactory` interface to create the uninitialized role and the `CosCompoundExternalization::internalize_role()` operation to internalize the role. The new roles should be co-located with the factory finder given by the `there` parameter.

The result of an `internalize_node()` operation is a sequence of roles.

Figure 5-6 illustrates the result of an `internalize()`. A **Node**, when it is born, is not in any relationships with other objects. That is, the roles in the new **Node** are disconnected. It is the `read_graph()` operation's job to correctly establish new relationships.

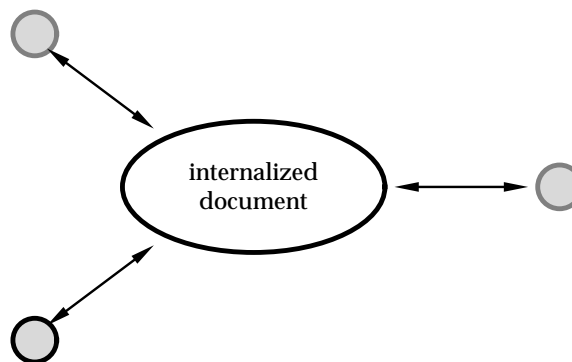


Figure 5-6 Internalizing a Node Returns New Object and Corresponding Roles

If an appropriate factory to internalize the roles cannot be found, the **NoFactory** exception is raised. The exception value indicates the key used to find the factory.

In addition to the **NoFactory** exception, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the internalized node, **NO_RESOURCES** will be raised.

5.7.2 The Role Interface

The **Role** interface defines operations to externalize and internalize a role. The **Role** interface also defines an operation to return the propagation value for the **externalize()** operation.

The implementation of a **CompoundExternalization::Node()** operation can call these operations on roles. For example, an implementation of externalize on a node can call the **externalize()** operation on the **Role**.

Externalizing a Role

```
void externalize_role (in ::CosStream::StreamIO sio);
```

The **externalize_role()** operation transfers the **Role**'s state to the **Stream sio**.

Internalizing a Role

```
void internalize_role (in ::CosStream::StreamIO sio);
```

The **internalize_role()** operation causes a **Role** to read its state from the **Stream** given by **sio**.

Getting a Propagation Value

```
::CosGraphs::PropagationValue externalize_propagation (
    in RelationshipHandle rel,
    in ::CosRelationships::RoleName toRoleName,
    out boolean sameForAll);
```

The **externalize_propagation()** operation returns the propagation value to the role **toRoleName** for the **externalize()** operation and the relationship **rel**. If the **Role** can guarantee that the propagation value is the same for all relationships in which it participates, **sameForAll** is true.

5.7.3 The Relationship Interface

The **Relationship** interface defines operations to externalize and internalize a relationship. The **Relationship** interface also defines an operation to return the propagation values for the **externalize()** operation.

Externalizing the Relationship

```
void externalize_relationship (
    in ::CosStream::StreamIO sio);
```

The **externalize_role()** operation transfers the **Role**'s state to the **Stream sio**.

Internalizing the Relationship

```
void internalize_relationship(
    in ::CosStream::StreamIO sio,
    in ::CosGraphs::NamedRoles newRoles);
```

The **internalize_relationship()** operation internalizes the state of a relationship from the **Stream** given by **sio**.

The values of the internalized relationship's attributes are defined by the implementation of this operation. However, the **named_roles** attribute of the newly created relationship must match

newRoles. That is, the internalized relationship relates objects represented by the **newRoles** parameter, not by the original relationship's named roles.

Getting a Propagation Value

```
::CosGraphs::PropagationValue externalize_propagation (  
    in::CosRelationships::RoleName fromRoleName,  
    in::CosRelationship::RoleName toRoleName,  
    out boolean sameForAll);
```

The **propagation_for()** operation returns the relationship's propagation value from the role **fromRoleName** to the role **toRoleName** for the **externalize()** operation. If the role named by **fromRoleName** can guarantee that the propagation value is the same for all relationships in which it participates, **SameForAll** is true.

5.7.4 The PropagationCriteriaFactory Interface

The **CosGraphs** module in the Relationship Service defines a general service for traversing a graph of related objects. The service accepts a call-back object supporting the **::CosGraphs::TraversalCriteria** interface. Given a node, this object defines which edges to emit and which nodes to visit next.

The **PropagationCriteriaFactory** creates a **TraversalCriteria** object that determines which edges to emit and which nodes to visit based on propagation values for the compound externalization operations.

Create a Traversal Criteria Based on Externalization Propagation

```
::CosGraphs::TraversalCriteria create_for_externalize( );
```

The **create()** operation returns a **TraversalCriteria** object for an operation **op()** that determines which edges to emit and which nodes to visit based on propagation values for **op()**. For a more detailed discussion see Section 6.4.2 on page 132.

5.8 Specific Externalization Relationships

The Relationship Service defines two important relationships: *containment* and *reference*. Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment is represented by a relationship with two roles: the **ContainsRole** and the **ContainedInRole**. Similarly, reference is represented by a relationship with two roles: **ReferencesRole** and **ReferencedByRole**.

Compound externalization adds externalization semantics to these specific relationships. That is, it defines propagation values for containment and reference.

5.9 The CosExternalizationContainment Module

The **CosExternalizationContainment** module defines the following interfaces:

- **Relationship** interface
- **ContainsRole** interface
- **ContainedInRole** interface.

```
#include <Containment.idl>
```

```
#include <CompoundExternalization.idl>
```

```
module CosExternalizationContainment {
```

```
    interface Relationship :
        ::CosCompoundExternalization::Relationship,
        ::CosContainment::Relationship {};
```

```
    interface ContainsRole :
        ::CosCompoundExternalization::Role,
        ::CosContainment::ContainsRole {};
```

```
    interface ContainedInRole :
        ::CosCompoundExternalization::Role,
        ::CosContainment::ContainedInRole {};
```

The **CosExternalizationContainment** module does not define new operations. It merely mixes in interfaces from the **CosCompoundExternalization** and **CosContainment** modules. Although it does not add any new operations, it refines the semantics of these operations.

The **CosExternalizationContainment::ContainsRole::propagation_for()** operation returns the following:

Operation	ContainsRole to ContainedInRole
externalize()	deep

The **CosExternalizationContainment::ContainedInRole::propagation_for** operation returns the following:

Operation	ContainedInRole to ContainsRole
externalize()	None.

The **CosRelationships::RoleFactory::create_role** operation will raise the **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosCompoundExternalization::Node** interface.

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosExternalizationContainment::ContainsRole** and **CosExternalizationContainment::ContainedInRole**. It will raise **MaxCardinalityExceeded** if the **CosExternalizationContainment::ContainedInRole** is already participating in a relationship.

5.10 The CosExternalizationReference Module

The **CosExternalizationReference** module defines these interfaces:

- **Relationship** interface
- **ReferencesRole** interface
- **ReferencedByRole** interface.

```
#include <Reference.idl>
#include <CompoundExternalization.idl>
```

```
module CosExternalizationReference {

    interface Relationship :
        ::CosCompoundExternalization::Relationship,
        ::CosReference::Relationship {};

    interface ReferencesRole :
        ::CosCompoundExternalization::Role,
        ::CosReference::ReferencesRole {};

    interface ReferencedByRole :
        ::CosCompoundExternalization::Role,
        ::CosReference::ReferencedByRole {};
};
```

The **CosExternalizationReference** module does not define new operations. It merely mixes in interfaces from the **CosCompoundExternalization** and **CosReference** modules. Although it does not add any new operations, it refines the semantics of these operations.

The **CosExternalizationReference::ReferencesRole::propagation_for()** operation returns the following:

Operation	ReferencesRole to ReferencedByRole
externalize()	None.

The **CosExternalizationReference::ReferencedByRole::propagation_for()** operation returns the following:

Operation	ReferencedByRole to ReferencesRole
externalize()	None.

The **CosRelationships::RoleFactory::create_role()** operation will raise the **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosCompoundExternalization::Node** interface.

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosExternalizationReference::ReferencesRole** and **CosExternalizationReference::ReferencedByRole**.

5.11 Standard Stream Data Format

An externalization client may create a **Stream** that supports a specific external representation data format that is intended to be portable across different CORBA implementations and on different CPU hardware. A client creates such a **Stream** object using a factory found by specifying a **Key** whose only **NameComponent** has a **NameComponent::id** whose value is the string literal **StandardExternalizationFormat**.

That format is described in this section.

Externalized Object Data

1 byte

tag byte = x'F0'	Key info	Object info
------------------	----------	-------------

A leading “tag” byte with a value of x'F0' marks the beginning of an object's externalized data. Following this is data associated with a **Key** that can be used to internalize the object. The **Key** information is then followed by the data written to the **StreamIO** for the object's state.

Key Info

1 byte

length = i	1st id string	2nd id string	...	i'th id string
------------	---------------	---------------	-----	----------------

The **Key** information consists of a byte containing an integer value, “i”, that indicates how many **Naming::NameComponents** make up the associated **Key**.

This byte is followed by “i” null-terminated sequences of char values that represent the **Naming::NameComponent::id** values for the **Key**. These values correspond to the C mapping of a CORBA string type. The **NameComponent::kind** values are not stored in this external data format.

Object Info

1 byte

tag byte	data value	tag byte	data value	...
----------	------------	----------	------------	-----

The object information is the sequence of bytes generated for one or more **write_<type>()** operations. For each **write_<type>()** operation, a single “tag” byte identifying the type of the primitive data is followed by the data. The tag byte gives the internalization implementation enough information to skip past object state for objects that cannot be created; for example, when a compatible implementation cannot be found on the internalizing ORB.

The tag byte values and data formats for each type are as indicated below for basic CORBA data types:

Tag	CORBA Type	Data Format
x'F1'	Char	one byte
x'F2'	Octet	one byte
x'F3'	Unsigned Long	four bytes, big_endian format
x'F4'	Unsigned Short	two bytes, big_endian format
x'F5'	Long	four bytes, big_endian format
x'F6'	Short	two bytes, big_endian format
x'F7'	Float	four bytes, IEEE 754 single precision format, sign bit first byte
x'F8'	Double	eight bytes, IEEE 754 single precision format, sign bit first byte
x'F9'	Boolean	TRUE => one byte == 1, FALSE => one byte == 0
x'FA'	String	null terminated sequence of bytes

5.11.1 Externalized Repeated Reference Data

1 byte 4 bytes

x'04'	Object number
-------	---------------

This format is used only when multiple objects reference the same object. Instead of storing the referenced object multiple times, the duplicate reference objects are stored in this format. Note that the object is represented by a long object number which indicates that the object has been stored already.

5.11.2 Externalized NIL Data

1 byte

x'05'

This is a special format used to indicate that there is no object stored in the **Stream**.

Relationship Service Specification

6.1 Service Description

Distributed objects are frequently used to model entities in the real world. As such, distributed objects do not exist in isolation. They are related to other objects.

Consider some examples of real-world entities and relationships:

- A person *owns* cars; a car is *owned* by one or more persons.
- A company *employs* one or more persons; a person is *employed* by one or more companies.
- A document *contains* figures; a figure is *contained in* a document.
- A document *references* a book; a book is *referenced by* one or more documents.
- A person *checks out* books from libraries. A library *checks out* books to people. A book is *checked out* by a person from a library.

These examples demonstrate several relationships:

- ownership relationships between people and cars
- employment relationships between companies and people
- containment relationships between documents and figures
- reference relationships between books and documents
- check out relationships between people, books and libraries.

Such relationships can be characterised along a number of dimensions:

- Type

Related entities and the relationships themselves are typed. In the examples, *employment* is a relationship defined between *people* and *companies*. The type of the relationship constrains the types of entities in the relationship; a company cannot employ a monkey since a monkey is not a person. Furthermore, employment is distinct from other relationships between people and companies.

- Roles of Entities in Relationships

A relationship is defined by a set of roles that entities have. In an employment relationship, a company plays an *employer* role and a person plays an *employee* role.

A single entity can have different roles in distinct relationships. Note that a person can play the owner role in an ownership relationship and the employee role in an employment relationship.

- Degree

Degree refers to the number of required roles in a relationship. The check out relationship is a ternary relationship; it has three roles: the borrower role, the lender role and the material role. A person plays the borrower role, a library plays the lender role, and a book plays the material role. Ownership, employment, containment and reference, on the other hand, are degree 2, or binary relationships.

- Cardinality

For each role in a relationship type, the maximum cardinality specifies the maximum number of relationships that may involve that role.

The containment relationship is a many-to-one relationship; a document contains many figures; a figure is contained in exactly one document. A many-to-many relationship is between two sets of entities. The ownership example is a many-to-many relationship; a person can own multiple cars; a car can have multiple owners. The check out relationship is a many-to-one-to-many relationship. A person can check out many books from many libraries. A book is checked out by one person from one library and a library can loan many books to many people.

- Relationship Semantics

Relationships often have relationship-specific semantics; that is, they define operations and attributes. For example, *job title* is an attribute of the employment relationship, while it is not an attribute of an ownership relationship. Similarly, *due date* is an attribute of the check out relationship.

For more discussion on object-oriented modelling and design with relationships, see the referenced *Object-oriented Modelling and Design*.

6.1.1 Key Features of the Relationship Service

- The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: *relationships* and *roles*. A role *represents* a CORBA object in a relationship. A relationship is created by passing a set of roles to a relationship factory.
- Relationships of arbitrary degree can be defined.
- Type and cardinality constraints can be expressed and checked. Exceptions are raised when cardinality and type constraints are violated. The Relationship Service does not define a new type system. Instead, the OMG IDL type system is used to represent relationship and role types. This allows the service to leverage CORBA solutions for type federation.
- The **Relationship** interface can be extended to add relationship-specific attributes and operations. Similarly, the **Role** interface can be extended to add role-specific attributes and operations.
- The Relationship Service defines three levels of service: *base*, *graph* and *specific*.
- The base level defines relationships and roles.
- When objects are related, they form graphs of related objects. The graph level extends the base level service with nodes and traversal objects. Traversal objects iterate through the edges of a graph. Traversals are useful in implementing compound operations on graphs, among other things.
- Specific relationships are defined by the third level.

A conforming Relationship Service implementation must implement level 1 or levels 1 and 2 or levels 1, 2 and 3.

- Appendix C, Life Cycle Operations on Distributed Object Graphs, of Common Object Services, Volume 1 defines operations to copy, move and remove graphs of related objects.
- The Relationship Service requires a notion of object identity. As such, it defines a simple, efficient mechanism for supporting object identity in a heterogeneous, CORBA-based

environment. This mechanism may be used for other services.

- Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references; role objects can be co-located with their objects and need not depend on a centralised repository of relationship information. As such, navigating a relationship can be a local operation.
- The Relationship Service allows so-called immutable objects to be related. There are no required interfaces that objects being related must support. As such, objects whose state and implementation were defined prior to the definition of the Relationship Service can be related objects.
- The Relationship Service allows graphs of related objects to be traversed without activating related objects.
- The Relationship Service is extensible. Programmers can define additional relationships.

6.1.2 The Relationship Service versus CORBA Object References

The CORBA Specification defines object references that clients use to issue requests on objects. Object references can be stored persistently. When is it appropriate to use object references and when is it appropriate to use the Relationship Service?

The Relationship Service is appropriate to use when an application needs any of the following capabilities that are not available with CORBA object references.

Relationships that are Multi-directional

When objects are related using the Relationship Service, the relationship can be navigated from any role to any other role. The service maintains the relationship between related objects. CORBA object references, on the other hand, are unidirectional. Objects that possess CORBA object references to each other can only do so in an *ad hoc* fashion; there is no way to maintain and manipulate the relationship between the objects.

Relationships that Allow Third-party Manipulation

Since roles and relationships are themselves CORBA objects, they can be exported to third parties. This allows third parties to manipulate the relationship. For example, a third party could create, destroy or navigate the relationship. Third parties cannot manipulate object references.

Traversals that are Supported for Graphs of Related Objects

When objects are related using the Relationship Service, they form graphs of related objects. Interfaces are defined by the Relationship Service to support traversing the graph.

Relationships and Roles that can be Extended with Attributes and Behaviour

Relationships have relationship-specific semantics. For example, the employment relationship has a *job title* attribute. Since relationships and roles are objects with well-defined OMG IDL interfaces, they can be extended through OMG IDL inheritance to add such relationship-specific attributes and operations.

6.1.3 Resolution of Technical Issues

Modelling and Relationship Semantics

An application designer models a problem as a set of objects and the relationships between those objects. Using OMG IDL, the application designer directly represents the objects of the model. Using the Relationship Service, the application designer directly represents the roles and relationships of the model.

The **Relationship** and **Role** interfaces can be extended using OMG IDL inheritance to add relationship and role-specific attributes and operations. For example, a designer might define the employment relationship to have an operation returning a job title.

Managing Relationships

The **RelationshipFactory** interface defines an operation to create a relationship, given a set of roles. The **Role** and **Relationship** interfaces define operations to delete and navigate relationships between objects.

Constraining Relationships

Type, cardinality and degree constraints on relationships are expressed in the interfaces.

The **RoleFactory::create_role** operation can raise a **RelatedObjectTypeError** exception. This allows implementations of the **Role** interface to place further constraints on the type of the related objects. For example, an **EmployedByRole** can ensure related objects are people. An attempt to have it represent a monkey would raise a **RelatedObjectTypeError** exception.

Similarly, the **RelationshipFactory::create()** operation can raise a **RoleTypeError** exception. This allows implementations of the **Relationship** interface to put constraints on the type of the roles. For example, an employment relationship can ensure there is an **EmployerRole** and an **EmployeeRole**.

The **RelationshipFactory::create()** operation can also raise a **DegreeError** exception. This ensures that there are the correct number of roles.

Maximum cardinality constraints are enforced by the role objects themselves. A role can raise a **MaxCardinalityExceeded** exception and refuse to participate in a relationship if its maximum cardinality would be exceeded. Roles define an operation to ask if their minimum cardinality constraint is being met.

Referential Integrity

If the Relationship Service is used in an environment supporting transactions, strict referential integrity is achieved. That is, if a related object refers to another (via a relationship), then the other related object will also refer to it. Without transactions, strict referential integrity cannot be achieved since a failure during execution of the relationship construction protocol could cause a dangling reference.

Relationships and Roles as First Class Objects

Our design defines both relationships and roles as first class objects. This is extremely important because it encapsulates and abstracts the state to represent the relationship, allows third-party manipulation of the relationship, and allows the roles and relationships themselves to support operations and attributes.

Different Models for Navigating and Constructing Relationships

The Relationship Service defines interfaces for constructing and navigating relationships component-by-component. These building block operations can be used by a higher-level service, such as a query service.

Efficiency Considerations

Our design has several features that allow for highly optimised implementations. Performance optimisations are achieved by clustering and/or caching of connection information.

Clients can cluster related objects and their roles by their selection of factories.

Our design defines the containment relationship logically. It does not imply physical clustering of state or execution. However, it serves as a good hint to implementations for clustering. An environment can choose to cluster containers and contained objects.

The `get_other_related_object()` operation can be implemented to *cache* remote related objects. The cached information is immutable; once a relationship is established, the roles and related objects will not change.

6.2 Service Structure

This section provides information about the levels of service; the specification is organised around these levels. It also describes the hierarchy of Relationship Service interfaces and explains the main purpose of each interface.

6.2.1 Levels of Service

The Relationship Service defines three levels of service: *base relationships*, *graphs of related objects* and *specific relationships*. The specification is organised around these levels.

Level One: Base Relationships

The **Relationship** and **Role** interfaces define the base Relationship Service. Figure 6-1 illustrates two instances of the containment relationship. The document plays the container role; the figure and the logo play the containee role.

The diamond is an object supporting the Relationship interface. The small circles are objects supporting the Role interface.

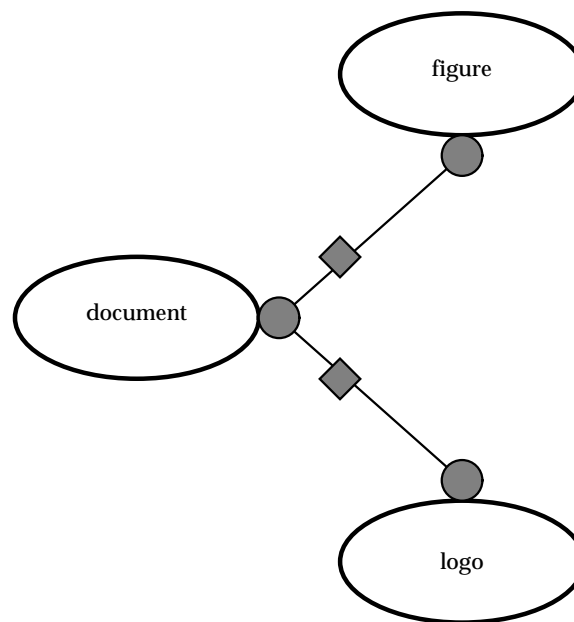


Figure 6-1 Base Relationships

Roles represent objects in relationships. Roles have a maximum cardinality. As illustrated, the container role can be involved in many instances of a relationship. The containee roles can only be involved in a single instance of a relationship.

Figure 6-2 on page 109 illustrates the navigation functionality of relationships; for example, the arrow between a role and another role indicates it is possible to navigate from one role to another. The arrow does not, however, indicate that the object reference to the other role is necessarily stored by the role.

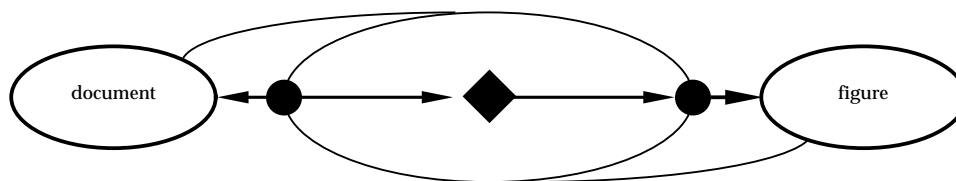


Figure 6-2 Navigation Functionality of Base Relationships

Figure 6-2 lists the interfaces to support relationships and roles. Section 6.3 on page 114 specifies the interfaces in detail.

Level Two: Graphs of Related Objects

Distributed objects do not exist in isolation. They are connected together. Objects connected together form graphs of related objects. The Relationship Service defines the **Traversal** interface. The **Traversal** interface defines an operation to traverse a graph. The traversal object cooperates with extended roles supporting the **CosGraphs::Role** interface and objects supporting the **Node** interface.

Figure 6-3 on page 110 illustrates a graph of related objects. The folder, the figure, the logo and the book all support the **Node** interface. The small circles are roles supporting the **CosGraphs::Role** interface.

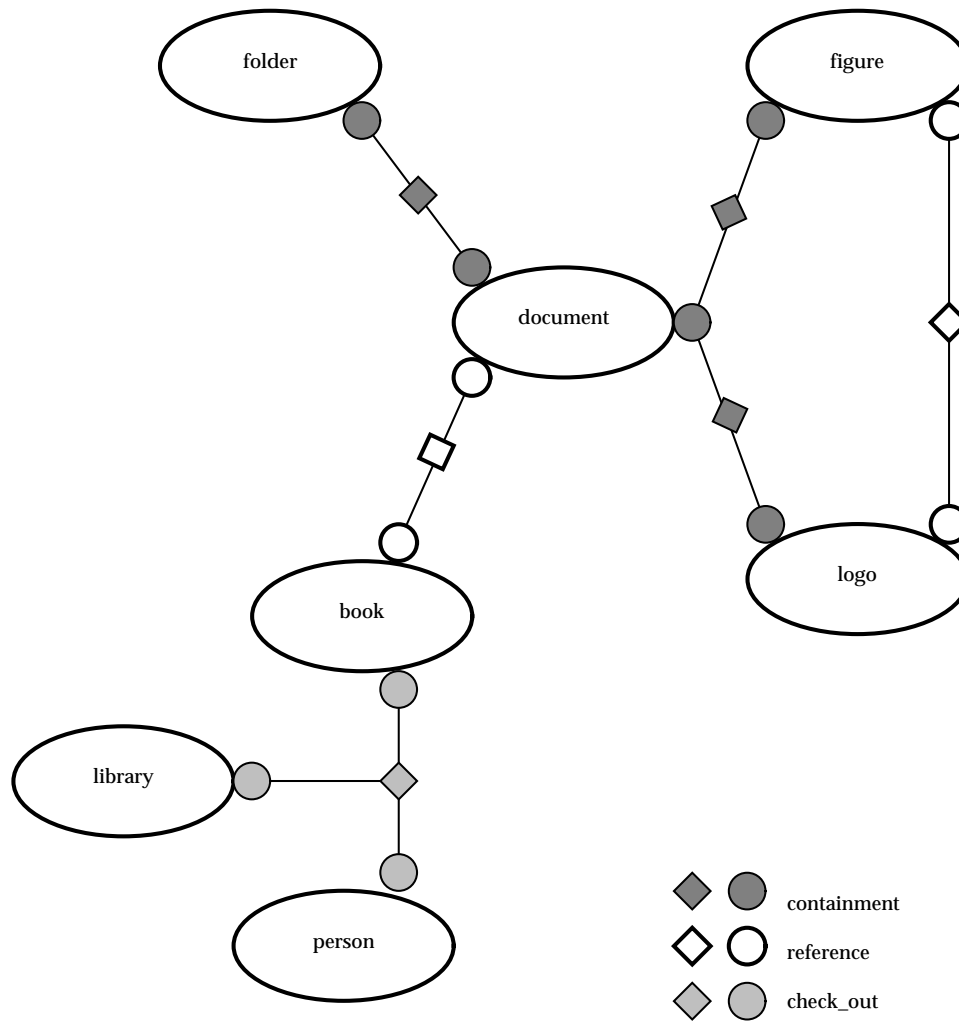


Figure 6-3 Example Graph of Related Objects

Figure 6-3 lists the interfaces to support graphs of related objects. Section 6.4 on page 130 specifies the interfaces in detail.

Level Three: Specific Relationships

Containment and reference are two important relationships. The Relationship Service defines these two binary relationships.

Figure 6-4 on page 111 and Figure 6-5 on page 111 list the interfaces defining specific relationships. Section 6.5 on page 142 specifies the interfaces in detail.

6.2.2 Hierarchy of Relationship Interface

The **Relationship** interfaces are arranged into the interface hierarchy illustrated in Figure 6-4.

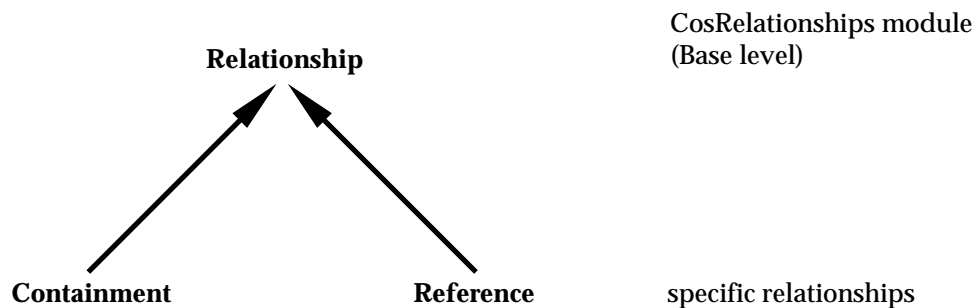


Figure 6-4 Relationship Interface Hierarchy

6.2.3 Hierarchy of Role Interface

The **Role** interfaces are arranged into the interface hierarchy illustrated in Figure 6-5.

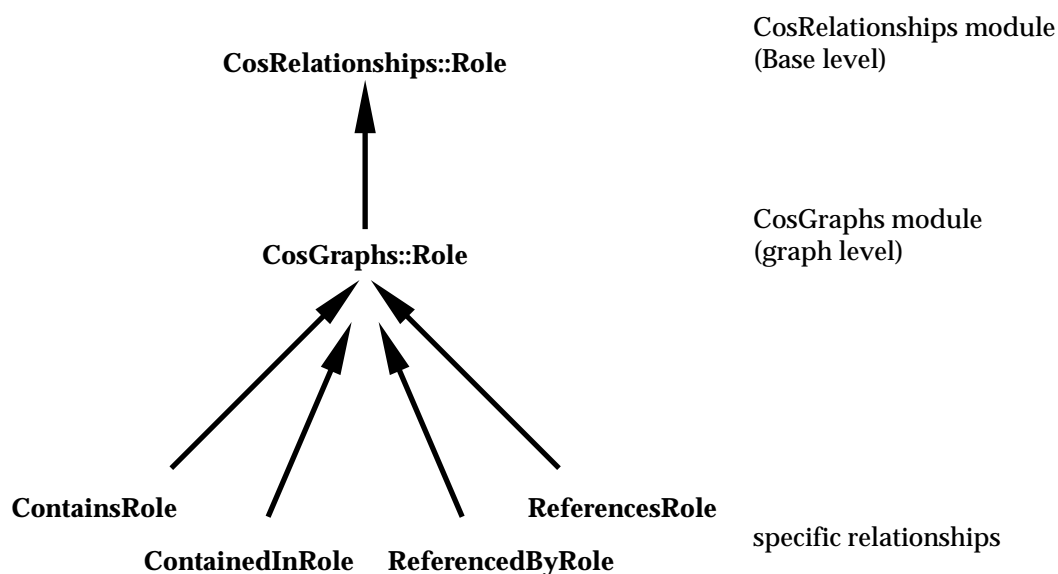


Figure 6-5 Role Interface Hierarchy

The **Role** interface defines operations to efficiently navigate relationships between related objects.

The **CosGraphs::Role** interface defines an operation to return the edges that involve the role. This is used by the traversal service defined at the graph level.

Finally, **ContainsRole**, **ContainedInRole**, **ReferencesRole** and **ReferencedByRole** are specific roles for two important relationships: *containment* and *reference*.

6.2.4 Interface Summary

The Relationship Service defines interfaces to support the functionality described in Section 6.2 on page 108.

Table 6-1 through Table 6-5 on page 113 give high-level descriptions of the Relationship Service interfaces. This chapter also describes the interfaces in detail.

Interface	Purpose	Primary Clients
CosObjectIdentity::IdentifiableObject	To determine whether two objects are identical.	There are many clients. The graph level of the Relationship Service is one.

Table 6-1 Interfaces Defined in the CosObjectIdentity Module

Interface	Purpose	Primary Clients
CosRelationships::Relationship	Represents an instance of a relation type.	Clients that navigate between related objects.
RelationshipFactory	Supports the creation of relationships.	Clients establishing relationships.
Role	Defines navigation operations for relationships. Implements type and cardinality constraints.	Clients that navigate between related objects. Relationship factories.
RoleFactory	Supports the creation of roles.	Objects participating in relationships.
RelationshipIterator	Iterates the relationships in which a particular role object participates.	Clients that navigate relationships.

Table 6-2 Interfaces Defined in the CosRelationship Module

Interface	Purpose	Primary Clients
CosGraphs::Traversal	Defines an operation to traverse a graph, given a starting node and traversal criteria.	Clients that want a standard service to traverse graphs.
TraversalFactory	Supports the creation of a traversal object.	Clients that want a standard service to traverse graphs.
TraversalCriteria	Provides navigation behaviour between nodes.	Traversal implementations.
Role	Extends the CosRelationships::Role interface to return edges.	Clients that traverse graphs of related objects.
EdgeIterator	Returns additional edges from a role.	Clients that traverse graphs of related objects.
Node	Defines operations for a related object to reveal its roles.	Clients that traverse graphs of related objects.
NodeFactory	Supports the creation of nodes.	Clients that create nodes in graphs.

Table 6-3 Interfaces Defined in the CosGraph Module

Interface	Purpose	Primary Clients
CosReferences::Relationship	Many-to-many relationship.	Clients that depend on reference relationship type.
ReferencesRole	Represents an object that references other objects.	Clients that navigate reference relationships between objects.
ReferencedByRole	Represents an object that is referenced by other objects.	Clients that navigate reference relationships between objects.

Table 6-4 Interfaces Defined in the CosReferences Module

Interface	Purpose	Primary Clients
CosContainments::Relationship	One-to-many relationship.	Clients that depend on containment relationship type.
ContainsRole	Represents an object that contains other objects.	Clients that navigate containment relationships between objects.
ContainedInRole	Represents an object that is contained in other objects.	Clients that navigate containment relationships between objects.

Table 6-5 Interfaces Defined in the CosContainments Module

6.3 The Base Relationship Model

The base level of the Relationship Service defines interfaces that support relationships between two or more CORBA objects. Objects that participate in a relationship are called *related objects*. Relationships that share the same semantics form *relationship types*. A relationship is an instance of a relationship type and has an identity.

Each related object is connected with the relationship via a role. Roles are objects which characterise a related object's participation in a relationship type. Role types are used for expressing the role's characteristics by an OMG IDL interface. Cardinality represents the number of relationship instances connected to a role. Degree represents the number of roles in a relationship. All characteristics are expressed by corresponding OMG IDL interfaces. Relationship and role types are built by subtyping the **Relationship** and **Role** interfaces.

Figure 6-6 gives a graphical representation of a simple relationship type. It illustrates that documents reference books. Documents are in the **ReferencesRole** and books are in the **ReferencedByRole**. Documents, reference, the roles and books are all types; there are interfaces (written in OMG IDL) for all five.

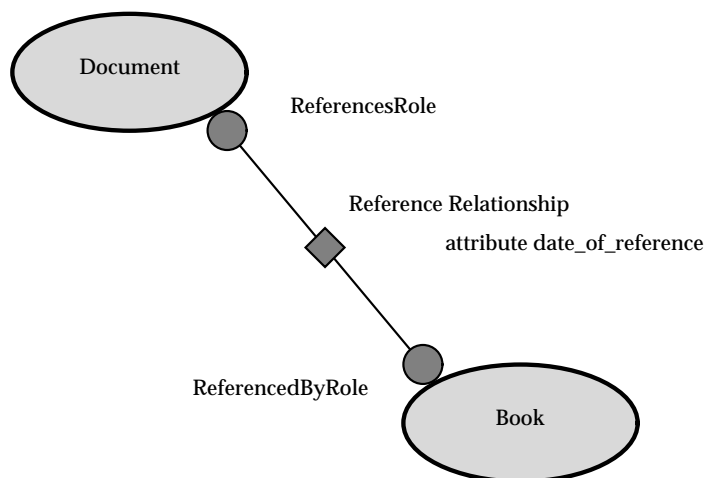


Figure 6-6 Simple Relationship Type: Documents Reference Books

Figure 6-7 on page 115, on the other hand, gives a graphical representation of an instance of a relationship type. It illustrates that my document, an instance of **Document**, references **War and Peace**, an instance of **Book**.⁵

5. Most of the figures in this specification represent instances of related objects, roles and relationships. Figures describing object and relationship type are clearly marked.

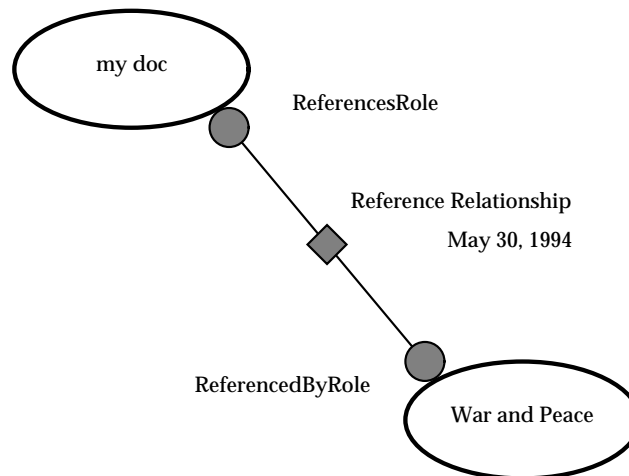


Figure 6-7 Simple Relationship Instance: My Document References the Book War and Peace

6.3.1 Relationship Attributes and Operations

Relationships may have attributes and operations. For example, the reference relationship of Figure 6-6 on page 114 has an attribute indicating the date the reference from the document to the book was established.

Rationale for Relationship Attributes and Operations

If relationships are not allowed to define attributes and operations, they will have to be assigned to one of the related objects. This approach is prone to misunderstandings and inconsistencies. The approach to define an artificial related object, which then carries the attributes, is equally unsatisfactory.

The **date** attribute of the example of Figure 6-7 is clearly an attribute of the relationship, not one of related objects. It cannot be an attribute of **my document** since **my document** can reference many books on different dates. Similarly, it cannot be an attribute of **War and Peace** since **War and Peace** can be referenced by many books on different dates.

6.3.2 Higher Degree Relationships

The reference relationship in Figure 6-6 on page 114 is a *binary* relationship; that is, it is defined by two roles. The Relationship Service can also support relationships with more than two roles. The fact that three or more related objects may be part of a relationship can be expressed directly by means of the same concept as in the binary case. The *degree* represents the number of roles in a relationship. The Relationship Service supports higher degree relationships; that is, relationships with degree greater than two.

Figure 6-8 on page 116 shows a ternary “check out” relationship between books, libraries and persons. The semantics of this relationship are that a person borrows a book from a library. The relationship also defines an attribute that indicates the date when the book is due to be returned by the person to the library.

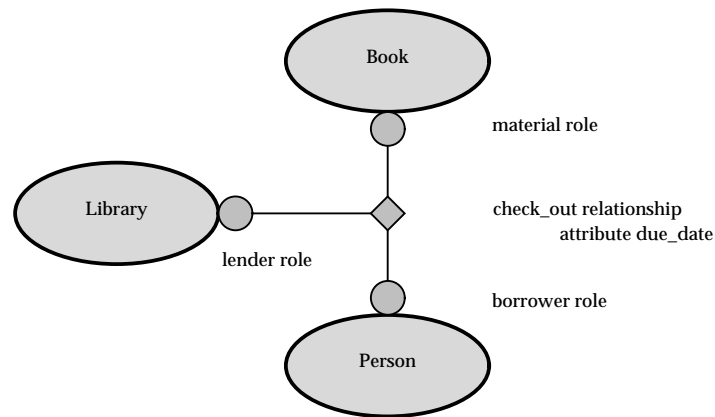


Figure 6-8 Satisfactory Ternary Check-out Relationship

Rationale for Supporting Higher Degree Relationships

The Relationship Service represents higher degree relationships directly. It clearly defines the number of expected related objects as well as other integrity constraints. It is more readable, more understandable and easier to enforce consistency constraints for related objects with a direct representation than with alternative representations that simulate higher degree relationships using a set of binary relationships. When simulating higher degree relationships, the relationship information is spread over multiple object and relationship type definitions, as are the corresponding integrity constraints.

Figure 6-9 shows an alternative representation of the ternary relationship from Figure 6-8 using binary relationships. Note that the first representation is not equivalent to that of Figure 6-8 since cardinalities and other integrity constraints cannot be expressed correctly in this alternative representation.

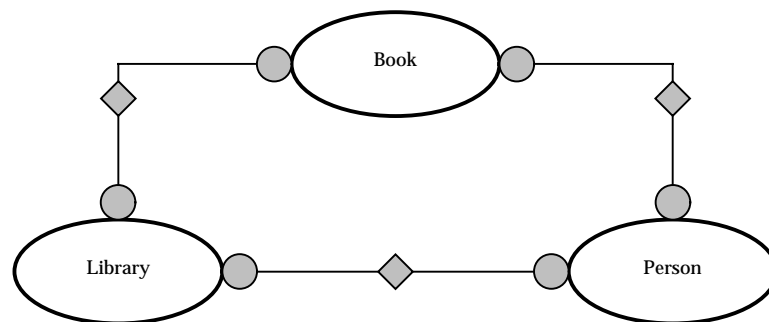


Figure 6-9 Unsatisfactory Ternary Check-out Relationship

Figure 6-10 on page 117 illustrates a second alternative representation of the ternary relationship of Figure 6-8. It uses an additional (artificial) related object type. This representation is equivalent to Figure 6-8 if **check_out** is constrained to participate in exactly one instance of each of the three binary relationship types. However, this alternative needs three relationship types and one additional related object type (**check_out**) instead of only one relationship type, and therefore is much more complex and harder to capture when compared to the representation using one relationship type with degree 3.

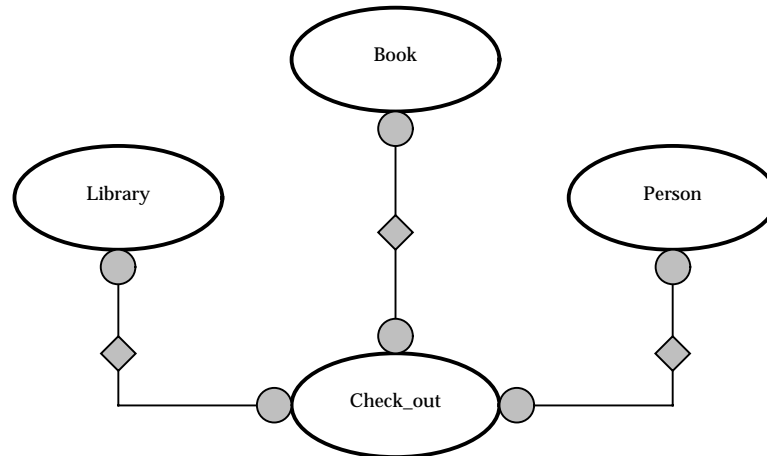


Figure 6-10 Another Unsatisfactory Representation

Since the Relationship Service supports higher order relationships directly, the user of the service need not resort to the unsatisfactory representations using binary relationships of Figure 6-9 on page 116 and Figure 6-10.

6.3.3 Operations

The base-level Relationship Service provides operations to:

- create role and relationship objects
- navigate relationships
- destroy roles and relationships
- iterate over the relationships in which a role participates.

Creation

Roles are constructed independently using a role factory. Roles represent an existing related object that is passed as a parameter to the **RoleFactory::create()** operation. When creating a new role object, the type of the related object can be checked by the factory. The minimum and maximum cardinality (for example, the minimal and the maximal number of relationship instances to which the new role object may be connected) are indicated by attributes on the factory.

Figure 6-11 illustrates a newly created role.

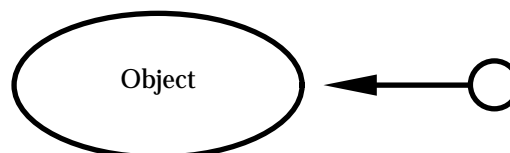


Figure 6-11 Creating a Role for an Object

A new relationship is created by passing a sequence of named roles to a factory for the relationship. The expected degree and role types for the new relationship are indicated by

attributes on the factory. During the creation of the new relationship, the role types and the maximum cardinality can be checked. Duplicate role names are not allowed since the names are used to distinguish the roles in the scope of the relationship.

When creating a relationship, the factory creates “links” between the roles and the relationship using the `link()` operation on the role.

Figure 6-12 illustrates a fully established binary relationship.⁶

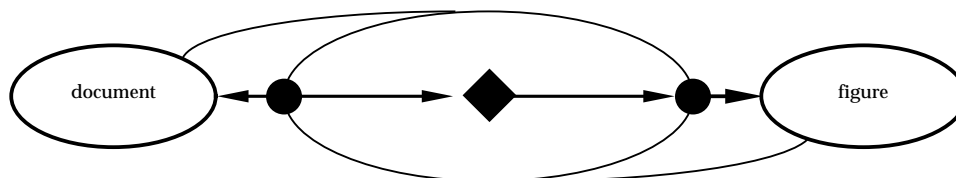


Figure 6-12 Fully Established Binary Relationship

Navigation

Figure 6-12 illustrates the navigational functionality of a relationship. In particular:

- A relationship defines an attribute that indicates a read-only attribute that indicates the named roles of the relationship.
- A role defines a read-only attribute that indicates the related object that the role represents.
- A role supports the `get_other_role()` operation, that given a relationship object and a role name, returns the other role object.
- A role supports the `get_other_related_object()` operation, that given a relationship object and a role name, returns the related object that the named role represents in the relationship.
- A role supports the `get_relationships()` operation which returns the relationships in which the role participates.

Destruction

For both roles and relationship objects, the Relationship Services introduces a `destroy()` operation. The `destroy()` operation for relationship objects also destroys the links between the relationship and all of the role objects.

6. Figure 6-12 represents *navigational functionality*; it does not necessarily represent stored object references. A variety of implementation strategies are described in Section 6.3.5 on page 119.

6.3.4 Consistency Constraints

For each role, two cardinalities are defined: *minimum* and *maximum*.

- The minimum cardinality indicates the minimum number of relationship instances in which a role must participate.
- The maximum cardinality indicates the maximum number of relationship instances in which a role can participate.

Maximum cardinality constraint can be checked when relationships are created. Note that the relationship mechanism cannot, by itself, enforce the minimum cardinality constraint. However, a role can be asked explicitly whether it meets its minimum cardinality constraint using the `check_minimum_cardinality()` operation.

Type integrity is preserved by CORBA mechanisms because related objects, roles and relationships are instances of CORBA object types. Type constraints can be checked when roles and relationships are created.

6.3.5 Implementation Strategies

Figure 6-12 on page 118 illustrates the navigational functionality of a fully established binary relationship. There are a variety of implementation strategies possible. The `get_other_role()` and the `get_other_related_object()` operations can be:

- implemented by caching object references to other roles and related objects
- computed when needed using the relationship object.

The appropriate implementation strategy typically depends on distribution boundaries. If the roles and relationship objects are clustered, then only storing the values at the relationship object optimises space. If, on the other hand, the roles and the related objects are clustered, caching object references to other roles and related objects at the roles allows the relationship to be efficiently navigated without involving a remote relationship object.

Role implementations that cache object references to other roles and related objects need not worry about updating the cache. Once the related objects and relationships are established, they cannot be changed.

6.3.6 The CosObjectIdentity Module

The CORBA Specification does not define a notion of object identity for objects. The Relationship Service requires object identity for the objects it defines. As such, the Relationship Service assumes the **CosObjectIdentity** module specified below. This is defined in a separate module; other object services may find this module to be generally useful.

```
module CosObjectIdentity {
    typedef unsigned long ObjectIdentifier;

    interface IdentifiableObject {
        readonly attribute ObjectIdentifier constant_random_id;
        boolean is_identical (
            in IdentifiableObject other_object);
    };
};
```

6.3.6.1 The IdentifiableObject Interface

Objects that support the **IdentifiableObject** interface implement an attribute of type **ObjectIdentifier** and the **is_identical()** operation. This mechanism provides an efficient and convenient method of supporting object identity in a heterogeneous CORBA-based environment.

constant_random_id

readonly attribute ObjectIdentifier constant_random_id;

Objects supporting the **IdentifiableObject** interface define an attribute of type **ObjectIdentifier**. The value of the attribute must not change during the lifetime of the object.

A typical client use of this attribute is as a key in a hash table. As such, the more randomly distributed the values are, the better.

The value of this attribute is not guaranteed to be unique; that is, another identifiable object can return the same value. However, if objects return different identifiers, clients can determine that two identifiable objects are *not* identical.

To determine whether two identifiable objects *are* identical, the **is_identical()** operation must be used.

is_identical()

boolean is_identical (
 in IdentifiableObject other_object);

The **is_identical()** operation returns **True** if the object and the **other_object** are identical. Otherwise, the operation returns **False**.

6.3.7 The CosRelationships Module

The **CosRelationships** module defines the interfaces of the base-level Relationship Service. In particular, it defines:

- **Relationship** and **Role** interfaces to represent relationships and roles
- **RelationshipFactory** and **RoleFactory** interfaces to create relationships and roles
- **RelationshipIterator** interface to enumerate the relationships in which a role participates.

The **CosRelationships** module is shown below:

```
#include <ObjectIdentity.idl>

module CosRelationships {

    interface RoleFactory;
    interface RelationshipFactory;
interface Relationship; interface Role;
    interface RelationshipIterator;

    typedef Object RelatedObject;
    typedef sequence<Role> Roles;
    typedef string RoleName;
    typedef sequence<RoleName> RoleNames;

    struct NamedRole {RoleName name; Role aRole;};
    typedef sequence<NamedRole> NamedRoles;
}
```

```

struct RelationshipHandle {
    Relationship the_relationship;

    CosObjectIdentity::ObjectIdentifier constant_random_id;
};
typedef sequence<RelationshipHandle> RelationshipHandles;

interface RelationshipFactory {
    struct NamedRoleType {
        RoleName name;
        ::CORBA::InterfaceDef named_role_type;
    };
    typedef sequence<NamedRoleType> NamedRoleTypes;
    readonly attribute ::CORBA::InterfaceDef relationship_type;
    readonly attribute
unsigned short degree;
    readonly attribute NamedRoleTypes named_role_types;
    exception RoleTypeError {NamedRoles culprits;};
    exception MaxCardinalityExceeded {
        NamedRoles culprits;};
    exception DegreeError {unsigned short required_degree;};
    exception
DuplicateRoleName {NamedRoles culprits;};
    exception UnknownRoleName {NamedRoles culprits;};

    Relationship create (in NamedRoles named_roles)
        raises (RoleTypeError,
            MaxCardinalityExceeded,
            DegreeError,
            DuplicateRoleName,
            UnknownRoleName);
};
interface Relationship :
    CosObjectIdentity::IdentifiableObject {
    exception CannotUnlink {
        Roles offending_roles;
    };
    readonly attribute NamedRoles named_roles;
    void destroy () raises(CannotUnlink);
};
interface Role { exception UnknownRoleName {};
    exception UnknownRelationship {};
    exception RelationshipTypeError {};
    exception CannotDestroyRelationship {

        RelationshipHandles offenders;
    };
    exception ParticipatingInRelationship {

        RelationshipHandles the_relationships;
    };
    readonly attribute RelatedObject related_object;
    RelatedObject
get_other_related_object (
        in RelationshipHandle rel,
        in RoleName target_name)
        raises (UnknownRoleName,
            UnknownRelationship);
    Role get_other_role (in RelationshipHandle rel,

```

```

        in RoleName target_name)
        raises (UnknownRoleName,
              UnknownRelationship);
void get_relationships (
    in unsigned long how_many,
    out RelationshipHandles rels,
    out RelationshipIterator iterator);
void destroy_relationships(

raises(CannotDestroyRelationship);
void destroy() raises(ParticipatingInRelationship);
boolean check_minimum_cardinality ();
void link (in RelationshipHandle rel,
          in NamedRoles named_roles)

raises(RelationshipFactory::MaxCardinalityExceeded,

RelationshipTypeError);
void unlink (in RelationshipHandle rel)
    raises (UnknownRelationship);
};
interface RoleFactory {
    exception NilRelatedObject {};
    exception RelatedObjectTypeError {};
    readonly attribute ::CORBA::InterfaceDef role_type;
    readonly attribute unsigned long max_cardinality;
    readonly attribute unsigned long min_cardinality;
    readonly attribute sequence
        <::CORBA::InterfaceDef> related_object_types;
    Role create_role (in RelatedObject related_object)
        raises (NilRelatedObject, RelatedObjectTypeError);
};
interface RelationshipIterator {
    boolean next_one (out RelationshipHandle rel);
    boolean next_n (in unsigned long how_many,
                  out RelationshipHandles rels);
    void destroy ();
};
};

```

Example of Containment Relationships

The example given in Figure 6-13 on page 123 is referred to throughout the following sections to describe roles and relationships. It represents two binary one-to-many containment relationships between a document, a figure and a logo.

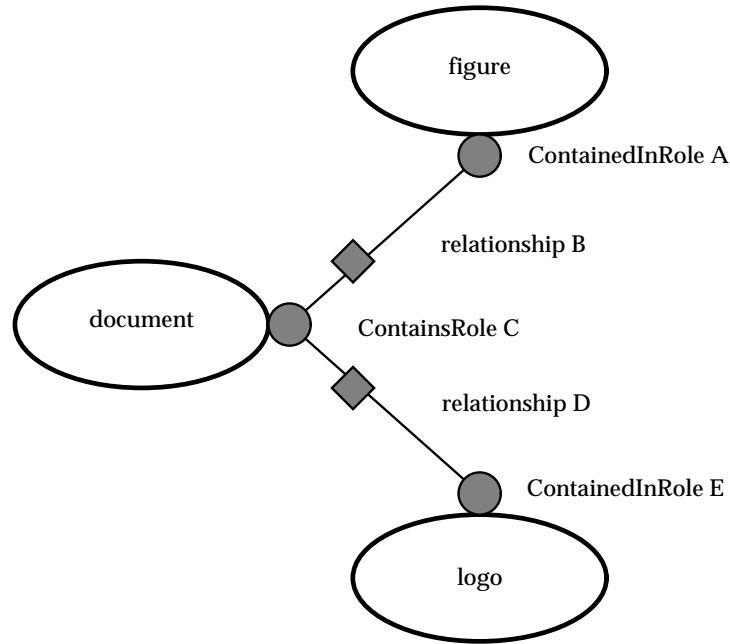


Figure 6-13 Two Binary One-to-many Containment Relationships

6.3.7.1 The RelationshipFactory Interface

The **RelationshipFactory** interface defines an operation for creating an instance of a relationship among a set of related objects. The factory also defines two attributes that specify the degree and role types of the relationships it creates.

Creating a Relationship

```
Relationship create (in NamedRoles named_roles)
    raises (RoleTypeError,
           MaxCardinalityExceeded,
           DegreeError, DuplicateRoleName,
           UnknownRoleName);
```

The **create()** operation creates a new instance of a relationship. The factory is passed a sequence of named roles that represent the related objects in the newly created relationship. The factory, in turn, informs the roles about the new relationship using the **link()** operation.

Roles implement maximum cardinality constraints. A role may refuse to participate in a new relationship because it would violate a cardinality constraint. In such a case, the **MaxCardinalityExceeded** exception is raised and the offending roles are returned in the exception.

The number of roles passed to the **create()** operation must be the same as the value of the **degree** attribute. If not, the **DegreeError** exception is raised.

Role names are used to associate each actual role object with one of the formal roles expected by the relationship to be created.

The set of role names passed to the **create()** operation must be the same as the set of role names in the factory's **named_role_types** attribute. If not, the **UnknownRoleName** exception is raised, and the unrecognised names are returned in the exception. The sequence order of the **named_roles** parameter and the sequence order of the **named_role_types** need not correspond.

The type of each role passed to the **create()** operation must be of the same type as the type indicated for the corresponding role name in the **named_role_types** attribute. If not, the **RoleTypeError** is raised and the offending roles are returned in the exception.

The names of the roles passed to the **create()** operation must be unique within the scope of this relationship type. If not, the **DuplicateRoleName** exception is raised.

In Figure 6-13 on page 123 the document and the figure were related; that is, relationship B was created by passing roles A and C to the **create()** operation of the relationship factory. Similarly, the document and the logo were related by passing roles C and E to the relationship factory for relationship D.

Determining the Created Relationship Type

readonly attribute ::CORBA::InterfaceDef relationship_type;

The relationship created by a factory may be a subtype of the **Relationship** interface. The **relationship_type** attribute indicates the actual types of the relationships created by the factory.

Determining the Degree of a Relationship Type

readonly attribute unsigned short degree;

The **degree** attribute indicates the number of roles for the relationships created by the factory.

For example, in Figure 6-13 on page 123 the relationship factory for containment has a **degree** attribute whose value is 2 because containment is a binary relationship.

Determining Names and Types of the Roles of a Relationship Type

readonly attribute NamedRoleTypes named_role_types;

The **named_role_types** attribute indicates the required names and types of roles for the relationships created by the factory. **NamedRoleTypes** are defined as structures where the role type is given by the **CORBA::InterfaceDef** for the role objects.

For example, in Figure 6-13 on page 123 the relationship factory for containment has an attribute whose value is a sequence of two **CORBA::InterfaceDefs**: one for **ContainsRole** and one for **ContainedInRole**.

6.3.7.2 The Relationship Interface

The **Relationship** interface defines an attribute whose value is the named roles of the relationship and an operation to destroy the relationship.

Determining the Roles of a Relationship and Their Names

readonly attribute NamedRoles named_roles;

The **named_roles** attribute returns the roles of the relationship. The roles have the names that were indicated in the **create()** operation defined by the **RelationshipFactory** interface.

For example, in Figure 6-13 on page 123 relationship B has an attribute whose value is a sequence:

```
<"A", InterfaceDef for ContainedInRole; "C", InterfaceDef for ContainsRole>
```

Similarly, relationship D has an attribute whose value is a sequence:

```
<"E", InterfaceDef for ContainedInRole; "C", InterfaceDef for ContainsRole>
```

Destroying a Relationship

void destroy () raises(CannotUnlink);

The **destroy()** operation destroys the relationship between the objects. The roles are unlinked by the relationship implementation before it is destroyed. If roles cannot be unlinked, the **CannotUnlink** exception is raised and the roles that could not be unlinked are returned in the exception.

For example, in Figure 6-13 on page 123 if **destroy()** is requested of relationship B, the **unlink()** operation is requested of both roles A and C and the relationship B is destroyed.

6.3.7.3 The Role Interface

The **Role** interface defines operations to:

- navigate the relationship from one role to another
- enumerate the relationships in which the role participates
- destroy all relationships in which the role participates
- link a role to a newly created relationship
- unlink a role in the destruction process of a relationship
- destroy the role itself.

Determining the Related Object that a Role Represents

readonly attribute RelatedObject related_object;

The **related_object** attribute indicates the related object that the role represents. The related object that the role represents is specified as a parameter to the **create()** operation defined by the **RoleFactory** interface.

Getting Another Related Object

```
RelatedObject get_other_related_object (
    in RelationshipHandle rel,
    in RoleName target_name)
raises (UnknownRoleName,
    UnknownRelationship);
```

The **get_other_related_object()** operation navigates the relationship **rel** to the related object represented by the role named **target_name**.

If the role does not know about a role named **target_name**, the **UnknownRoleName** exception is raised. If the role does not know about the relationship **rel**, the **UnknownRelationship** exception is raised.

For example, in Figure 6-13 on page 123, assuming role A is named "A", requesting **get_other_related_object(B, "A")** of role C returns the figure. On the other hand, requesting **get_other_related_object(D, "E")** of role C returns the logo.

Getting Another Role

Role **get_other_role** (in RelationshipHandle rel,
in RoleName target_name)
raises (UnknownRoleName, UnknownRelationship);

The **get_other_role** () operation navigates the relationship **rel** to the role named **target_name**. The role is returned.

If the role does not know about a role named **target_name** for the relationship **rel**, the **UnknownRoleName** exception is raised. If the role does not know about the relationship **rel**, the **UnknownRelationship** exception is raised.

For example, in Figure 6-13 on page 123, assuming role A is named "A", requesting **get_other_role(B,"A")** of role C returns role A. On the other hand, requesting **get_other_role(D,"E")** of role C returns role E.

Getting All Relationships in which a Role Participates

void **get_relationships** (
in unsigned long how_many,
out RelationshipHandles rels,
out RelationshipIterator iterator);

The **get_relationships** () operation returns the relationships in which the role participates.

The size of the list is determined by the **how_many** argument. If there are more relationships than specified by the **how_many** argument, an iterator is created and returned with the additional relationships. If there are no more relationships, a nil object reference is returned for the iterator. (The **RelationshipIterator** interface is a standard iterator described in the next section.)

For example, in Figure 6-13 on page 123, requesting **get_relationships** () on role C would return the relationships B and D.

Destroying All Relationships in which a Role Participates

void **destroy_relationships** (
raises(CannotDestroyRelationship);

The **destroy_relationships** () operation destroys all relationships in which the role participates.

The **destroy_relationships** () operation is semantically equivalent to requesting destroy of each relationship in which the role participates. The operation is not required to be implemented in that fashion.

If the **destroy_relationships** () operation cannot destroy one of the relationships, then the **CannotDestroyRelationship** exception is raised and the relationships that could not be destroyed are returned in the exception.

For example, in Figure 6-13 on page 123 requesting **destroy_relationships** () of role A causes relationship B to be destroyed. On the other hand, requesting **destroy_relationships** () of role C causes relationships B and D to be destroyed.

Destroying a Role

```
void destroy() raises(ParticipatingInRelationship);
```

The **destroy()** operation destroys the role. The role must not be participating in any relationships. If it is, the **ParticipatingInRelationship** exception is raised and the relationships in which the role participates are returned in the exception.

For example, in Figure 6-13 on page 123 requesting **destroy_role()** of role A destroys relationship B and role A.

Checking Minimum Cardinality of a Role

```
boolean check_minimum_cardinality ();
```

The **check_minimum_cardinality()** operation returns **true** if a role satisfies its minimum cardinality constraints. Otherwise, the operation returns **false**.

For example, in Figure 6-13 on page 123 requesting **check_minimum_cardinality()** of role A would return **true** since it is participating in relationship B.

Linking a Role in a Newly Created Relationship

```
void link (in RelationshipHandle rel,  
           in NamedRoles named_roles)  
raises(RelationshipFactory::MaxCardinalityExceeded,  
       RelationshipTypeError);
```

Note: The **link()** operation is not intended for general-purpose clients that create, navigate and destroy relationships. Instead, it is an operation intended for implementations of the relationship factory **create()** operation.

The **link()** operation informs the role that a new relationship is being created. The role is passed a relationship and a set of named roles that represent related objects in the relationship.

A role can have a maximum cardinality; that is, it may limit the number of relationships in which it participates. If the **link()** request would cause the maximum to be exceeded, the **MaxCardinalityExceeded** exception is raised. If the type of the relationship does not agree with the relationship type that the role expects, the **RelationshipTypeError** exception is raised.

For example, in Figure 6-13 on page 123, when creating relationship B, the factory for B requested the **link(B,A,C)** operation on roles A and C. This allows roles A and C to support the navigation and administration operations for relationship B.

Removing a Role from a Relationship

```
void unlink (in RelationshipHandle rel)  
             raises (UnknownRelationship);
```

Note: The **unlink()** operation is not intended for general-purpose clients that create, navigate and destroy relationships. Instead, it is an operation intended for implementations of the relationship **destroy()** operation.

The **unlink()** operation causes the role to delete its record of the relationship.

If the relationship passed as an argument is unknown to the role, the **UnknownRelationship** exception is raised.

For example, in Figure 6-13 on page 123 the implementation of the **destroy()** operation on relationship B requests **unlink(B)** of roles A and C. This causes roles A and C to forget their participation in relationship B.

6.3.7.4 The RoleFactory Interface

The **RoleFactory** interface defines attributes describing the roles that it creates and a single operation to create a role.

Creating a Role

Role create_role (in RelatedObject related_object)
raises (NilRelatedObject, RelatedObjectTypeError);

The **create_role()** operation creates a role for the related object passed as a parameter.

A role must represent a related object. If a nil object reference is passed to the factory for the related object, the **NilRelatedObject** exception is raised.

Role factories can restrict the type of objects the roles they create will represent. If the interface of the related object does not conform, the **RelatedObjectTypeError** exception is raised.

For example, in Figure 6-13 on page 123 clients that created roles A, C and E used the **create()** operation of factories that support the **RoleFactory** interface.

Determining the Created Role Type

readonly attribute ::CORBA::InterfaceDef role_type;

The role created by a factory may be a subtype of the **Role** interface. The **role_type** attribute indicates the actual types of the roles created by the factory.

Determining the Maximum Cardinality of a Role

readonly attribute unsigned long max_cardinality;

The **max_cardinality** attribute indicates the maximum number of relationships in which a role (created by the factory) participates.

For example, in Figure 6-13 on page 123 the factory for role A returns 1, since a **ContainedInrole** can be in no more than one relationship. Attempts to add role A to more than one relationship result in **MaxCardinalityExceeded** exceptions. (See the **create()** operation of the **RelationshipFactory** interface and the **link()** operation of the **Role** interface.)

Determining the Minimum Cardinality of a Role

readonly attribute unsigned long min_cardinality;

The **min_cardinality** attribute indicates the minimum number of relationships in which a role (created by the factory) participates.

Note that, unlike maximum cardinality, minimum cardinality cannot be enforced since roles will be below their minimum during relationship construction. Roles do support the **check_minimum_cardinality()** operation to report if they are below their minimum.

For example, in Figure 6-13 on page 123 the factory for role A returns 1, since a **ContainedIn** role should be in one relationship.

Determining the Related Object Types for a Role

readonly attribute sequence

<::CORBA::InterfaceDef> related_object_types;

The factory creates roles that represent related objects in relationships. The related objects must support at least one of the interfaces indicated by the **related_object_type** attribute.

For example, in Figure 6-13 on page 123 the factory for role C returns the **CORBA::InterfaceDef** for a document.

6.3.7.5 The RelationshipIterator Interface

The **RelationshipIterator** interface is returned by the **get_relationships()** operation defined by the **Role** interface. It allows clients to iterate through any additional relationships in which the role participates.

next_one()

boolean next_one (out RelationshipHandle rel);

The **next_one()** operation returns the next relationship; if no more relationships exist, it returns **false**.

next_n()

boolean next_n (in unsigned long how_many,
out RelationshipHandles rels);

The **next_n()** operation returns at most the requested number of relationships; if no more relationships exist, it returns **false**.

destroy()

void destroy ();

The **destroy()** operation destroys the iterator.

6.4 Graphs of Related Objects

When objects are related using the Relationship Service, graphs of related objects are formed. This section focuses on how the Relationship Service supports graphs of related objects. We first describe the graph architecture supported by the service, describe support for traversing the graph and implementing compound operations and then specify the **CosGraphs** module in detail.

Graphs are important for distributed, object-oriented applications. A few examples of graphs are:

- Distributed Desktops

Folders and objects are connected together. Folders contain some objects and reference others. Folders may contain or reference other folders. The objects are distributed; they span multiple machines. The distributed desktop is a distributed graph.

- Composed Applications

Applications are built out of existing objects that are connected together. An example of such a composed application is a shared white board. The composed application is a graph.

- User Interface Hierarchies

Presentation objects visualise semantic objects for users. Presentations contain other presentation objects. For example, a window might contain a button. The user interface hierarchy is a graph.

- Compound Documents

A compound document architecture allows graphics, animation, sound, video, and so on, to be connected together to give the user the impression of a single document. The compound document is a graph.

6.4.1 Graph Architecture

A graph is a set of nodes and a set of edges, involving those nodes. Nodes are related objects that support the Node interface and edges are represented by the relationships that relate nodes.

Figure 6-14 on page 131 illustrates an example of a graph.

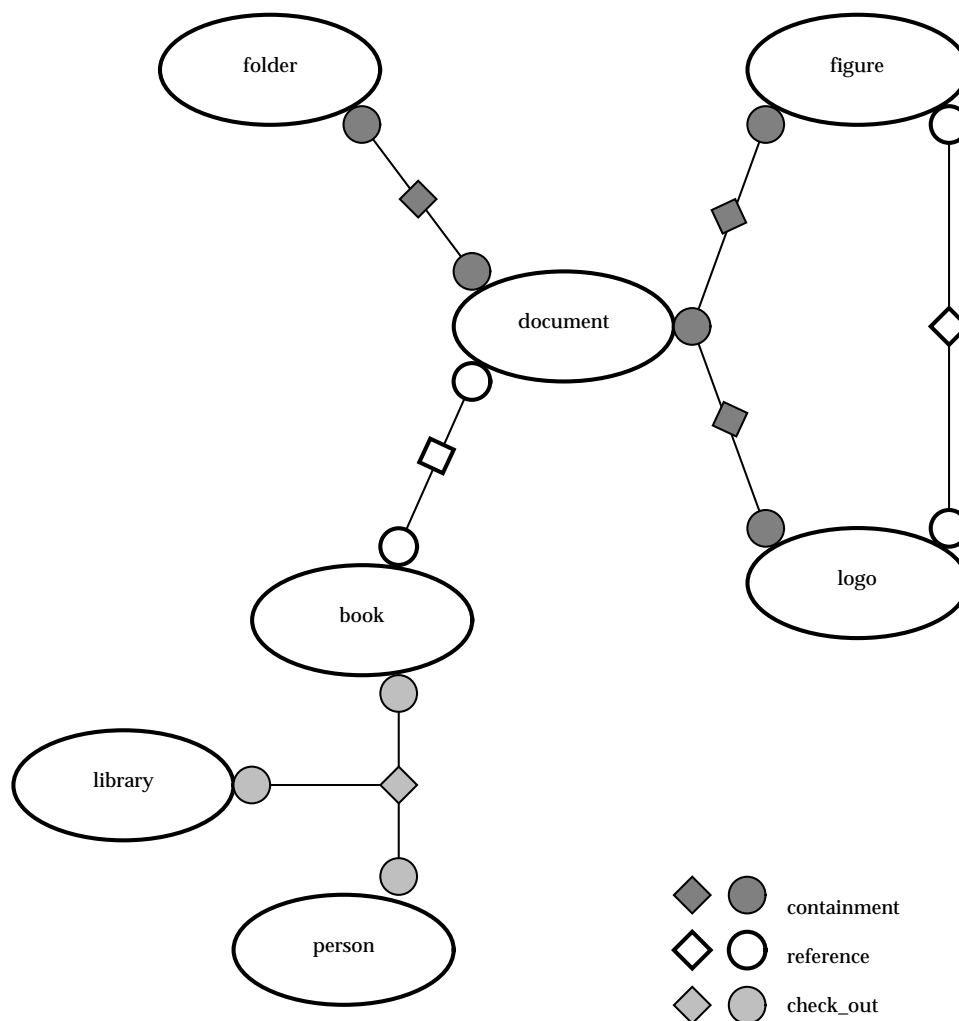


Figure 6-14 Example Graph of Related Objects

The folder, book, document, figure, library, person and logo are nodes in the graph. The edges of the graph are represented by the relationships:

- containment: the folder and document
- containment: the document and the figure
- containment: the document and the logo
- reference: the figure and the logo
- reference: the document and the book
- check_out: the book, the library and the person.

The graph architecture supports multiple kinds of relationships. For example, in Figure 6-14, there are containment, reference and check_out relationships. The small circles depict roles for a reference relationship, the solid circles depict roles for a containment relationship, and the shaded circles represent the roles of the check_out relationship.

A node can participate in more than one kind of relationship and thus have more than one role. In the example, the document has three kinds of role:

- the **ContainsRole**
- the **ContainedInRole**
- the **ReferencesRole**.

Nodes

Nodes are identifiable objects that support the **Node** interface. Nodes collect roles of a related object and the related object itself. A node enables standard traversals of graphs of related objects because it supports the following:

- a read-only attribute defining all of its roles
- an operation allowing roles of a particular type to be returned
- operations to add and remove roles.

The **Node** interface can be inherited by related objects, or an object implementing the **Node** interface can be instantiated and interposed in front of related objects. Interposition is particularly useful in these cases:

- when connecting immutable objects, which are objects that are not aware of the Relationship Service
- in order to traverse graphs of related objects without activating the related objects.

As such, the **Node** interface defines an attribute whose value is the related object it represents.

6.4.2 Traversing Graphs of Related Objects

The Relationship Service defines a traversal object that, given a starting node, produces a sequence of directed edges of the graph. A directed edge corresponds to a relationship. In particular, it consists of:

- an instance of a relationship
- a starting node and a starting named role of the edge to indicate direction
- a sequence containing the remaining nodes and named roles — for binary relationships, there is a single remaining node and role; for n-ary relationships, there are n-1 remaining nodes and roles.

The traversal object works like an iterator, where directed edges are the items being returned.

The traversal object, the nodes and the roles cooperate in traversing the graph. Through the operations of the **Node** interface, the node reveals its roles to the traversal object. Through the operations of the **CosGraphs::Role** interface, a role reveals its directed edges to other nodes. (The **CosGraphs::Role** interface defines an operation allowing a role to reveal directed edges.)

In traversing a graph, the traversal object must detect and represent cycles, and determine the relevant nodes and edges.

Detecting and Representing Cycles

In order to terminate, a traversal must be able to detect a cycle in the graph. In Figure 6-14 on page 131 the document, the figure and the logo form a cycle.

To detect cycles in the graph, the traversal object depends on the fact that nodes are identifiable objects, that is they support the **IdentifiableObject** interface defined in Section 6.3.6 on page 119.

To represent cycles in the graph, the traversal object defines a scope of identifiers for the nodes and relationships in the graph. That is, a given traversal assigns identifiers to the nodes and relationships that are guaranteed to be unique within the scope of the traversal.

Determining the Relevant Nodes and Edges

A traversal begins at the starting node, emits directed edges, and may continue to other related nodes. The traversal object is programmable in the criteria it uses for determining the edges to emit and the nodes to visit. The traversal object depends on a “call-back” object supporting the **TraversalCriteria** interface.

Given a node, the traversal criteria computes a sequence of directed edges to include in the traversal. For each edge, the traversal criteria can indicate whether the traversal should continue to an adjacent node. Based on the results of the traversal criteria, the traversal object emits edges and visits other nodes. The process continues until there are no more edges to emit and no more nodes to visit.

Three standard traversal modes are defined to allow clients flexibility in controlling the search order:

- depth first
- breadth first
- best first.

In order to understand the differences between the modes, consider that the traversal maintains an ordered list of the edges which have been produced by visiting nodes. This list initially contains the edges which result from visiting the root node. In each iteration the first edge is removed from the list to be returned and its destination nodes are visited. Depending upon the traversal mode, these edges are inserted in the beginning of the list (depth first), appended to the end of the list (breadth first), or inserted into the list which is sorted by the edge's weight (best first).

6.4.3 Compound Operations

Traversal objects are especially important in implementing compound operations on graphs of related objects. By compound operations, we mean operations that apply to some subset of the nodes and edges in the graph. Examples of compound operations include operations such as **copy**, **move**, **remove**, **externalize**, **print**, and so on.

Note: The Relationship Service defines a framework for compound operations but does not define specific compound operations. The Life Cycle and the Externalization Service specifications define compound operations that depend on the Relationship Service.

A compound operation may be implemented either in one or two passes. A compound operation implemented in one pass traverses the graph itself and applies the operation as it proceeds.

A compound operation implemented in two passes uses the traversal object defined by the Relationship Service to determine the relevant nodes and detect and represent cycles. The

second pass simply applies the operation to the results of the first pass.

A compound operation implemented in two passes provides a **TraversalCriteria** object for the traversal service.

6.4.4 An Example Traversal Criteria

Consider a traversal of a graph with a traversal criteria object that uses propagation values defined by the relationships to determine whether to emit an edge and whether to proceed to another node. The traversal criteria is given a node by the traversal. The traversal criteria then requests propagation values from each of the node's roles.

Figure 6-15 illustrates a traversal of a graph using a traversal criteria for a compound **copy()** operation. Using the **propagation_for()** operation defined by the **CompoundLifeCycle:Role** interface, the traversal criteria obtains the propagation value for the **copy()** operation from each of the node's roles.

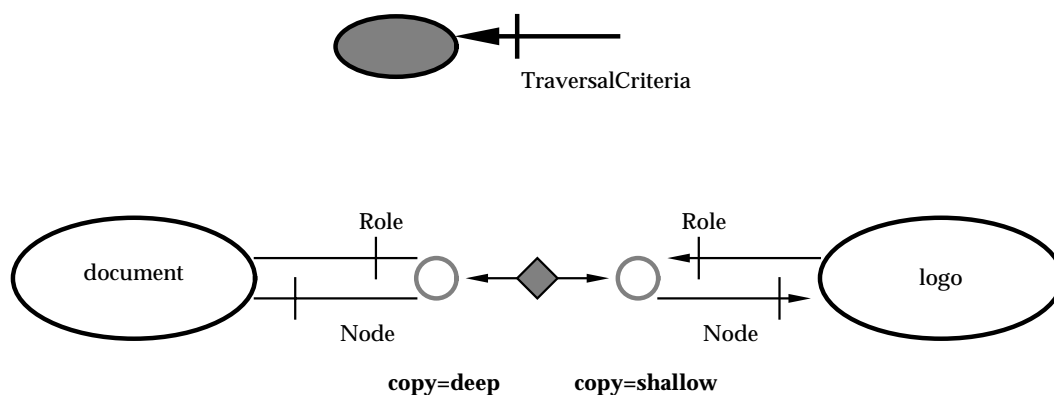


Figure 6-15 Traversal of a Graph for a Compound **copy()** Operation

Propagation

Compound operations may propagate from one node to another depending on the semantics of the relationship between the nodes. The propagation semantics of a relationship depend on the direction the relationship is being traversed. A propagation value is either **deep**, **shallow**, **inhibit** or **none**.

deep means that the operation is applied to the node, to the relationship and to the related objects. In the example of Figure 6-15, the propagation value for the **copy()** operation is **deep** from the document to the logo; the **copy()** propagates from the document to the logo across the containment relationship. The traversal criteria for **copy()** that encounters a **deep** propagation value would instruct the traversal object to emit the edge and visit the logo.

shallow means that the operation is applied to the relationship but not to the related objects. In Figure 6-15, the propagation value for the **copy()** operation from the logo to the document is **shallow**. The traversal criteria for **copy()** that encounters a **shallow** propagation value would instruct the traversal object to emit the edge but the document is not visited.

none means that the operation has no effect on the relationship and no effect on the related objects. A traversal criteria that encounters a **none** propagation value would not return any edges and related nodes are not visited.

Figure 6-16 summarises how **deep**, **shallow** and **node** propagation values affect nodes, roles and relationships.

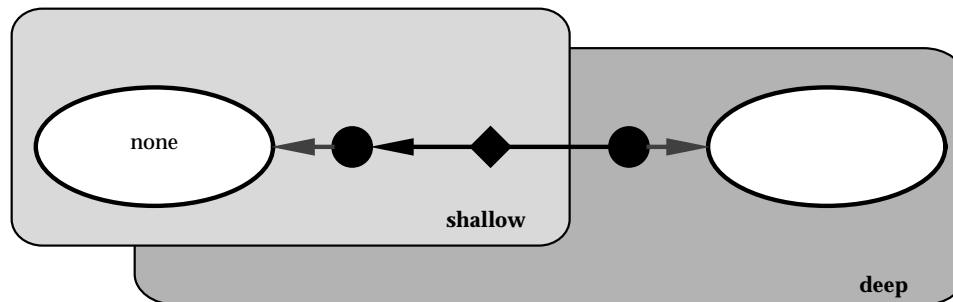


Figure 6-16 deep, shallow and none Propagation Values

inhibit means that the operation should not propagate to the node via any of the node's roles. **inhibit** is particularly meaningful for the **remove()** operation to provide so-called "existence-ensuring relationships".

For more discussion of propagation values, see the referenced *Controlling Propagation of Operations Using Attributes on Relations*.

6.4.5 The CosGraphs Module

The **CosGraphs** module defines the support for graphs of related objects. It defines the following interfaces:

- **TraversalFactory** interface for creating traversal objects
- **Traversal** interface for enumerating directed edges of a graph
- **TraversalCriteria** "call-back" interface to allow programmability of the traversal object
- **Node** interface for collecting the roles of a related object
- **NodeFactory** interface for creating nodes
- **Role** interface to support traversals.

The **CosGraphs** module is shown below:

```
#include <Relationships.idl>
#include <ObjectIdentity.idl>

module CosGraphs {

    interface TraversalFactory;
    interface Traversal;
    interface TraversalCriteria;
    interface Node;
    interface NodeFactory;
    interface Role;
    interface Edgelterator;

    struct NodeHandle {
        Node the_node;
        ::CosObjectIdentity::ObjectIdentifier constant_random_id;
    };
};
```

```

typedef sequence<NodeHandle> NodeHandles;

struct NamedRole {
    Role the_role;
    ::CosRelationships::RoleName the_name;
};
typedef sequence<NamedRole> NamedRoles;

struct EndPoint {
    NodeHandle the_node;
    NamedRole the_role;
};
typedef sequence<EndPoint> EndPoints;

struct Edge {
    EndPoint from;
    ::CosRelationships::RelationshipHandle the_relationship;
    EndPoints relatives;
};
typedef sequence<Edge> Edges;

enum PropagationValue {deep, shallow, none, inhibit};
enum Mode {depthFirst, breadthFirst, bestFirst};

interface TraversalFactory {
    Traversal create_traversal_on (
        in NodeHandle root_node,
        in TraversalCriteria the_criteria,
        in Mode how);
};

interface Traversal {
    typedef unsigned long TraversalScopedId;
    struct ScopedEndPoint {
        EndPoint point;
        TraversalScopedId id;
    };
    typedef sequence<ScopedEndPoint> ScopedEndPoints;
    struct ScopedRelationship {
        ::CosRelationships::RelationshipHandle
        scoped_relationship;
        TraversalScopedId id;
    };
    struct ScopedEdge {
        ScopedEndPoint from;
        ScopedRelationship the_relationship;
        ScopedEndPoints relatives;
    };
    typedef sequence<ScopedEdge> ScopedEdges;
    boolean next_one (out ScopedEdge the_edge);
    boolean next_n (in short how_many,
        out ScopedEdges the_edges);
    void destroy ();
};

interface TraversalCriteria {
    struct WeightedEdge {
        Edge the_edge;
        unsigned long weight;
        sequence<NodeHandle> next_nodes;
    };
};

```

```

        typedef sequence<WeightedEdge> WeightedEdges;
        void visit_node(in NodeHandle a_node,
                       in Mode search_mode);
        boolean next_one (out WeightedEdge the_edge);
        boolean next_n (in short how_many,
                      out WeightedEdges the_edges);
        void destroy();
    };
    interface Node : ::CosObjectIdentity::IdentifiableObject {
        typedef sequence<Role> Roles;
        exception NoSuchRole {};
        exception DuplicateRoleType {};

        readonly attribute ::CosRelationships::RelatedObject
            related_object;
        readonly attribute Roles roles_of_node;
        Roles roles_of_type (
            in ::CORBA::InterfaceDef role_type);
        void add_role (in Role a_role)
            raises (DuplicateRoleType);
        void remove_role (in ::CORBA::InterfaceDef of_type)
            raises (NoSuchRole);
    };

    interface NodeFactory {
        Node create_node (in Object related_object);
    };
    interface Role : ::CosRelationships::Role {
        void get_edges ( in long how_many,
                       out Edges the_edges,
                       out Edgelterator the_rest);
    };
    interface Edgelterator {
        boolean next_one (out Edge the_edge);
        boolean next_n ( in unsigned long how_many,
                       out Edges the_edges);
        void destroy ();
    };
};

```

6.4.5.1 The TraversalFactory Interface

The **TraversalFactory** interface creates traversal objects. The **Traversal** interface is used by clients that want to traverse graphs of related objects according to some traversal criteria.

```

create_traversal_on()
Traversal create_traversal_on (
    in NodeHandle root_node,
    in TraversalCriteria the_criteria,
    in Mode how);

```

The **create_traversal_on()** operation creates a traversal object starting at the **root_node**. The created traversal object uses the **TraversalCriteria** object to determine which directed edges to emit and which nodes to visit. The mode parameter indicates whether the traversal will proceed in a depth first, breadth first or best first fashion.

6.4.5.2 *The Traversal Interface*

Traversal objects iterate through **ScopedEdges** of the graph according to the traversal criteria and the mode established when the traversal was created. The traversal also defines a scope for the nodes and edges it returns; that is, it assigns identifiers to the nodes and edges it returns. The identifiers are unique within the scope of a given traversal. **ScopedEdges** are given by the following structure:

```
struct ScopedEdge {
    ScopedEndPoint from;
    ScopedRelationship the_relationship;
    ScopedEndPoints relatives;
};
typedef sequence<ScopedEdge> ScopedEdges;
```

A **ScopedEdge** consists of a distinguished scoped end point, a scoped relationship and a sequence of scoped end points. The distinguished scoped end point indicates the direction of the edge. The scoped end point consists of a node, a role and an identifier for the node that is unique within the scope of the traversal.

next_one()

```
boolean next_one (out ScopedEdge the_edge);
```

The **next_one()** operation returns the next scoped edge; if no more scoped edges exist, it returns false.

next_n()

```
boolean next_n (in short how_many,
               out ScopedEdges the_edges);
```

The **next_n()** operation returns at most the requested number of scoped edges.

destroy()

```
void destroy ();
```

The **destroy()** operation destroys the traversal.

6.4.5.3 *The TraversalCriteria Interface*

The **TraversalCriteria** interface is used by the traversal object to determine which edges to emit and which nodes to visit from a given node. The traversal criteria behaves like an iterator of weighted edges. Weighted edges are given by the following structure:

```
struct WeightedEdge {
    Edge the_edge;
    unsigned long weight;
    sequence<NodeHandle> next_nodes;
};
typedef sequence<WeightedEdge> WeightedEdges;
```

A **WeightedEdge** consists of an edge, a weight and a sequence of nodes indicating whether the traversal should continue to the nodes. The weight is only meaningful for the best first traversal.

next_one()

boolean next_one (out WeightedEdge the_edge);

The **next_one()** operation returns the next weighted edge; if no more weighted edges exist, it returns **false**.

next_n()

**boolean next_n (in short how_many,
out WeightedEdges the_edges);**

The **next_n()** operation returns at most the requested number of weighted directed edges.

destroy()

void destroy();

The **destroy()** operation destroys the traversal criteria.

visit_node()

**void visit_node(in NodeHandle a_node,
in Mode search_mode);**

The **visit_node()** operation establishes the node for which the traversal criteria will iterate and indicates the current search mode. As the traversal object traverses the graph, it visits nodes by requesting the **visit_node()** operation of the traversal criteria, followed by **next_one()** or **next_n()** requests to obtain the outgoing edges from the node.

For **depthFirst** and **breadthFirst** modes, the weight field in the weighted edges is ignored. In the **bestFirst** mode, the weight value is utilised to order the traversal's edges list which is sorted by this value in ascending order.

If weighted edges from a previous node remain when **visit_node()** is requested, the traversal criteria discards the previous edges.

6.4.5.4 The Node Interface

The **Node** interface defines operations that are useful in navigating graphs of related objects. In particular, it defines:

- a read-only attribute giving all of the node's roles
- an operation allowing roles conforming to a particular type to be returned
- operations to add and remove roles.

Roles are distinguished in nodes in the OMG IDL of their interfaces.

A node cannot possess two roles where one role is a subtype of the other. This is precluded by the **add_role()** operation.

A node can possess two or more roles that have a common supertype. The set of roles can be obtained by passing the common supertype to the **roles_of_type()** operation.

related_object

```
readonly attribute ::CosRelationships::RelatedObject
    related_object;
```

The **related_object** attribute gives the related object that the node represents. This is useful when relating immutable objects.

roles_of_node

```
readonly attribute Roles roles_of_node;
```

The **roles_of_node** attribute gives all of the node's roles.

roles_of_type()

```
Roles roles_of_type (
    in ::CORBA::InterfaceDef role_type);
```

The **roles_of_type()** operation returns the node's roles that conform to the **role_type** parameter. A role conforms to **role_type** if it's interface is the same or is a subtype of **role_type**.

add_role()

```
void add_role (in Role a_role)
    raises (DuplicateRoleType);
```

The **add_role()** operation adds a role to the node. If the node possesses a role of the same type, a supertype or a subtype of **a_role**, the **DuplicateRoleType** exception is raised.

remove_role()

```
void remove_role (in ::CORBA::InterfaceDef of_type)
    raises (NoSuchRole);
```

The **remove_role()** operation removes all the roles that conform to the **of_type** parameter. If no roles conform to the **of_type** parameter, the **NoSuchRole** exception is raised.

6.4.5.5 The NodeFactory Interface

The **NodeFactory** interface defines a single operation for creating nodes.

create_node()

```
Node create_node (in Object related_object);
```

The **create_node()** operation creates a node whose **related_object** attribute is initialized to the **related_object** parameter.

6.4.5.6 The Role Interface

The **CosGraphs::Role** interface extends the **CosRelationships::Role** interface with a single operation to return a role's view of it's relationships. The role's view of a relationship is given by the following **Edge** structure:

```
struct Edge {
    EndPoint from;
    ::CosRelationships::RelationshipHandle the_relationship;
    EndPoints relatives;
};
typedef sequence<Edge> Edges;
```

The **Edge** structure is defined by an end point, a relationship and the other end points. The **from** end point is the role and its related object.

get_edges()

```
void get_edges ( in long how_many,  
                out Edges the_edges,  
                out Edgiterator the_rest);
```

The **get_edges()** operation returns the edges in which the role participates.

The size of the list is determined by the **how_many** argument. If there are more edges than specified by the **how_many** argument, an iterator is created and returned. If there are no more edges, a nil object reference is returned for the iterator.

6.4.5.7 *The Edgiterator Interface*

The **Edgiterator** interface is returned by the **get_edges()** operation defined by the **CosGraphs::Role** interface. It allows clients to iterate through any additional relationships in which the role participates.

next_one()

```
boolean next_one (out Edge the_edge);
```

The **next_one()** operation returns the next edge; if no more edges exist, it returns **false**.

next_n()

```
boolean next_n (in unsigned long how_many,  
               out Edges the_edges);
```

The **next_n()** operation returns at most the requested number of edges.

destroy()

```
void destroy ();
```

The **destroy()** operation destroys the iterator.

6.5 Specific Relationships

The Relationship Service defines two important relationships: *containment* and *reference*. The example used throughout this specification has been written in terms of these two relationships.

6.5.1 Containment and Reference

Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container.

Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment and reference are examples of relationships. However, since containment and reference are very common relationships, the Relationship Service defines them as standard.

Containment is defined by interfaces for a relationship and two roles:

- the **CosContainment::Relationship** interface
- the **CosContainment::ContainsRole** interface
- the **CosContainment::ContainedInRole** interface.

Relationship is a subtype of **CosRelationships::Relationship**, and **ContainedInRole** and **ContainsRole** are subtypes of **CosGraphs::Role**.

Similarly, reference is defined by interfaces for a relationship and two roles:

- the **CosReference::Relationship** interface
- the **CosReference::ReferencesRole** interface
- the **CosReference::ReferencedByRole** interface.

Relationship is a subtype of **CosRelationships::Relationship**, and **ReferencesRole** and **ReferencedByRole** are subtypes of **CosGraphs::Role**.

6.5.2 The CosContainment Module

The **CosContainment** module is given below:

```
#include <Graphs.idl>

module CosContainment {

    interface Relationship :
        ::CosRelationships::Relationship {};

    interface ContainsRole : ::CosGraphs::Role {};

    interface ContainedInRole : ::CosGraphs::Role {};

};
```

The **CosContainment** module does not define new operations. It introduces new OMG IDL types to represent containment. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory Attribute	Value
relationship_type	CosContainment::Relationship
degree	2
named_role_types	"ContainsRole", CosContainment::ContainsRole; "ContainedInRole", CosContainment::ContainedInRole

The `CosRelationships::RelationshipFactory::create()` operation will raise `DegreeError` if the number of roles passed as arguments is not 2. It will raise `RoleTypeError` if the roles are not `CosContainment::ContainsRole` and `CosContainment::ContainedInRole`. It will raise `MaxCardinalityExceeded` if the `CosContainment::ContainedInRole` is already participating in a relationship.

RoleFactory Attribute for ContainsRole	Value
role_type	CosContainment::ContainsRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosGraphs::Node

The `CosRelationships::RoleFactory::create_role()` operation will raise `RelatedObjectTypeError` if the related object passed as a parameter does not support the `CosGraphs::Node` interface. The `CosRelationships::RoleFactory::link()` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the `CosContainment::Relationship` interface.

RoleFactory Attribute for ContainedInRole	Value
role_type	CosContainment::ContainedInRole
maximum_cardinality	1
minimum_cardinality	1
related_object_types	CosGraphs::Node

The `CosRelationships::RoleFactory::create_role()` operation will raise `RelatedObjectTypeError` if the related object passed as a parameter does not support the `CosGraphs::Node` interface. The `CosRelationships::RoleFactory::link()` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the `CosContainment::Relationship` interface. The `CosRelationships::RoleFactory::link()` operation will raise `MaxCardinalityExceeded` if it is already participating in a containment relationship.

6.5.3 The CosReference Module

The `CosReference` module is given below.

```
#include <Graphs.idl>
```

```
module CosReference {
```

```
    interface Relationship :  
        ::CosRelationships::Relationship {};
```

```
    interface ReferencesRole : CosGraphs::Role {};
```

```
    interface ReferencedByRole : ::CosGraphs::Role {};
```

```
};
```

The **CosReference** module does not define new operations. It introduces new OMG IDL types to represent reference. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory Attribute	Value
relationship_type	CosReference::Relationship
degree	2
named_role_types	"ReferencesRole", CosReference::ReferencesRole ; "ReferencedByRole", CosReference::ReferencedByRole

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosReference::ReferencesRole** and **CosReference::ReferencedByRole**.

RoleFactory Attribute for ReferencesRole	Value
role_type	CosReference::ReferencesRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosGraphs::Node

The **CosRelationships::RoleFactory::create_role()** operation will raise **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosGraphs::Node** interface. The **CosRelationships::RoleFactory::link()** operation will raise **RelationshipTypeError** if the **rel** parameter does not conform to the **CosReference::Relationship** interface.

RoleFactory Attribute for ReferencedByRole	Value
role_type	CosReference::ReferencedByRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosGraphs::Node

The **CosRelationships::RoleFactory::create_role()** operation will raise **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosGraphs::Node** interface. The **CosRelationships::RoleFactory::link()** operation will raise **RelationshipTypeError** if the **rel** parameter does not conform to the **CosRelationship::Relationship** interface.

Transaction Service Specification

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 implementor
- an implementor's view of the Transaction Service, which will interest Transaction Service and ORB providers.

Appendix A on page 207 explains the relationship between the Transaction Service and TP standards.

7.1 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 centralised 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 together the transaction paradigm, essential to developing reliable distributed applications, and the object paradigm, key to productivity and quality in application development, to address the business problems of commercial transaction processing.

7.1.1 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 (non-catastrophic) 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 OODBMS 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.

7.1.2 Transactional Applications

The Transaction Service provides transaction synchronisation 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 7.4.1 on page 172). 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 **Explicit Propagation** on page 172).

The Transaction Service does not require that all requests are 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 behaviour when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

7.1.3 Definitions

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

- transactional client (TC)
- transactional objects (TO)
- recoverable objects
- transactional servers
- recoverable servers.

Figure 7-1 shows a simple application which includes these basic elements.

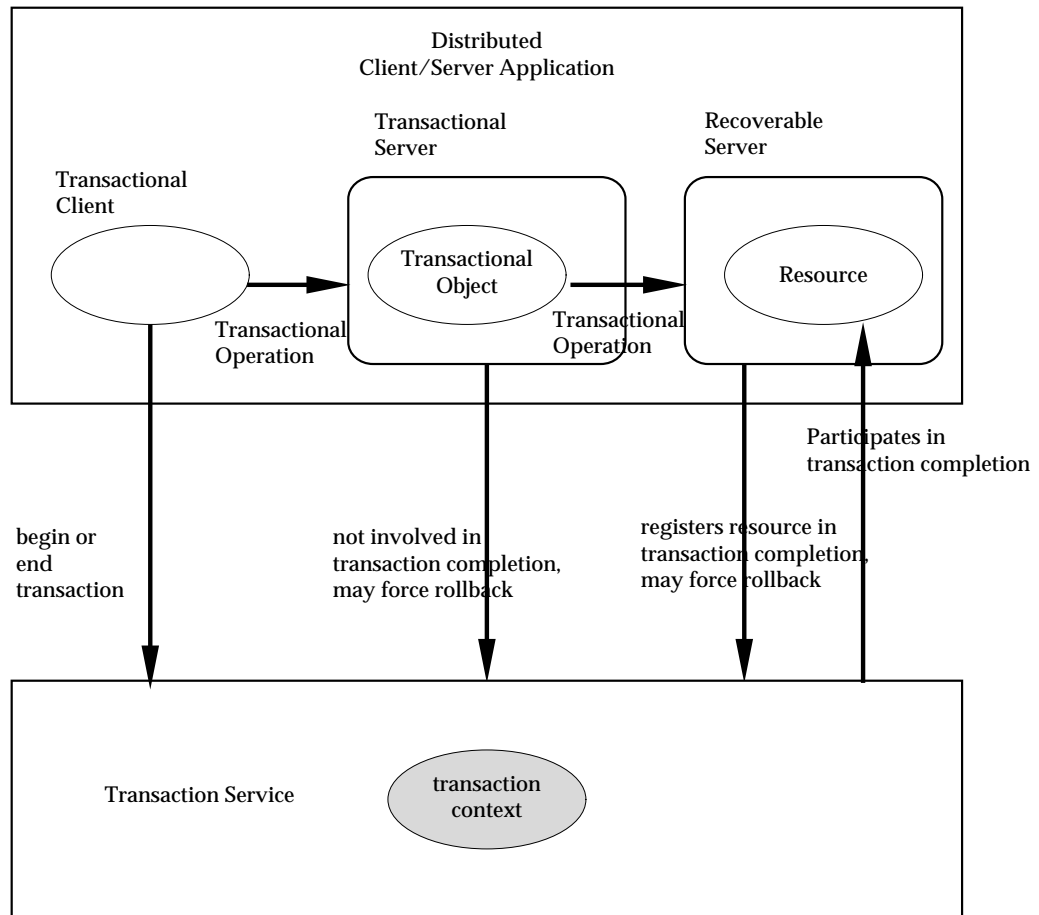


Figure 7-1 Application Including Definitions

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*.

Transactional Object

We use the term *transactional object* to refer to an object whose behaviour 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 behaviour, even when issued within the scope of a transaction. An object can choose to not support transactional behaviour, or to support transactional behaviour for some requests but not others.

We use the term *non-transactional 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 behaviour 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 behaviour 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 non-transactional implementations. No OMG IDL extensions are introduced to specify whether or not an operation has transactional behaviour. Transactional behaviour 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.

Recoverable Objects and Resource Objects

To implement transactional behaviour, 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.

Transactional Server

A *transactional server* is a collection of one or more objects whose behaviour 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.

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.

7.1.4 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.

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, are modelled on the X/Open Distributed TP Model.

A flat transaction is considered to be a top-level transaction (see **Nested Transactions**) 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.

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.

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.

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 Distributed TP Model. This is called *checked transaction behaviour*.

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 behaviour*.

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.

7.1.5 Principles of Function, Design and Performance

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

Functional Requirements

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

1. 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.

2. 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 re-implement all existing business functions — an essential element for commercial success in this marketplace.

3. 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 is 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 TP 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 design accommodates this requirement for implementations where interoperability with X/Open Distributed TP-compliant transactional applications is necessary.

4. 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 defined in this document 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. Requirements are identified in Section 7.5.2 on page 194.

5. 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 behaviour and objects without transactional behaviour within the execution of a single transaction.
- An object can specify transactional behaviour for its interfaces.

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

6. Support for TP Monitors

Customers require the ability to use object technology in building mission-critical applications. These applications are deployed on commercial transaction processing systems where a TP Monitor is used to provide 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 defined in this specification is usable in a TP Monitor environment.

Design Requirements

The Transaction Service defined by this specification supports the following design requirements:

1. Exploitation of OO Technology

The specification permits a wide variety of ORB and Transaction Service implementations and uses objects to enable ORB-based, secure implementations. The architecture of the Transaction Service is defined in terms of the CORBA Specification. All architected interactions among components are defined by OMG IDL and realised using the ORB. The ORB has sole responsibility for communication.

Consideration has been given to providing the programmer with simple, easy-to-use interfaces which hide some of the complexity inherent in the generality of the complete specification. Meaningful user applications can be constructed using a set of interfaces that are as simple or simpler than their procedural equivalents.

2. Low Implementation Cost

The Transaction Service specification has considered cost from the perspective of three users of the service — clients, ORB implementors 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 Manager implementations.

3. 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 individual ORB implementations.

4. 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 services interfaces can support transactional behaviour without change.

5. 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.

6. Wide Spectrum of Implementation Choices

The Transaction Service allows implementations to choose the degree of checking provided to guarantee legal behaviour of its users. This permits both robust implementations which provide strong assurances for transaction integrity and lightweight implementations where such checks are not warranted.

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 Distributed TP Model, in the following areas:

1. the number of network messages required
2. the number of disk accesses required
3. the amount of data logged.

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

7.2 Service Architecture

Figure 7-2 illustrates the major components and interfaces defined by the Transaction Service.

The transaction originator is an arbitrary program that begins a transaction. The recoverable server implements an object with recoverable state that is invoked within the scope of the transaction, either directly by the transaction originator or indirectly through one or more transactional objects.

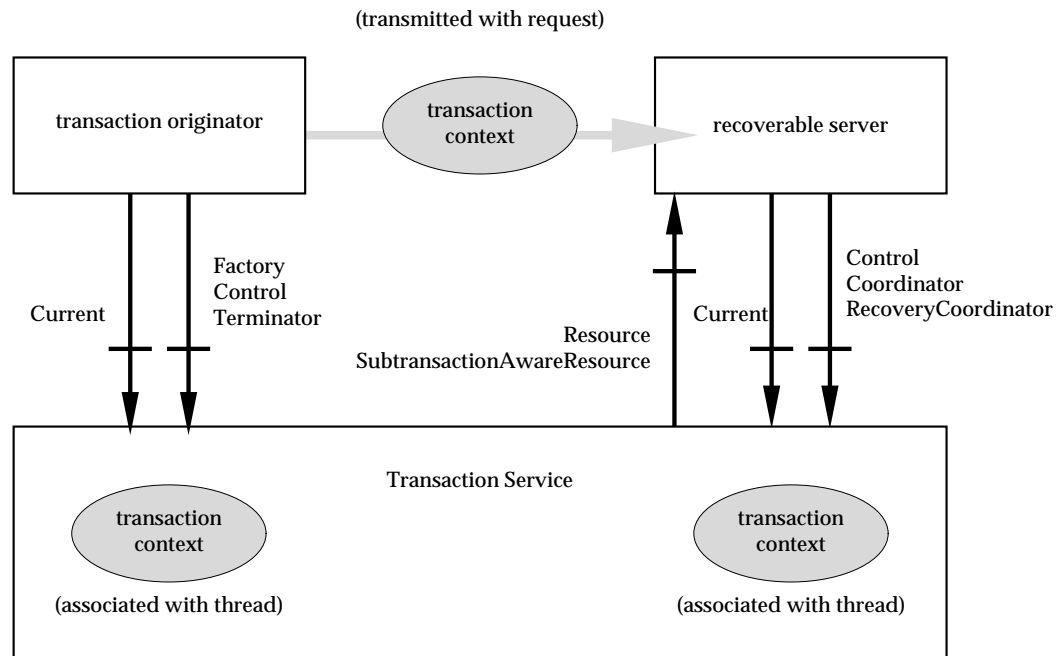


Figure 7-2 Major Components and Interfaces of the Transaction Service

The transaction originator creates a transaction using a **Factory**; 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 can also register a specialised 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.

7.2.1 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 roll back the transaction and can enquire 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 pseudo-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 is issued to the registered resources.

7.2.2 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 7.2.5 on page 158 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 (for example, 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.

7.2.3 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 pseudo-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.

7.2.4 Data Types

The **CosTransactions** module defines the following data types:

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

enum Vote {
    VoteCommit,
    VoteRollback,
    VoteReadOnly
};
```

7.2.5 Exceptions

7.2.5.1 Standard Exceptions

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

```
exception TransactionRequired {};  
exception TransactionRolledBack {};  
exception InvalidTransaction {};
```

These exceptions are standard exceptions, meaning that any request can raise one of these exceptions even though the exception is not (and must not be) declared in the operation signature.

TransactionRequired Standard Exception

Any operation can raise the **TransactionRequired** exception to indicate that the request carried a null transaction context, but an active transaction is required.

TransactionRolledBack Standard Exception

Any operation can raise the **TransactionRolledBack** exception to indicate that the transaction associated with the request has already been rolled back or marked to rollback; thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

InvalidTransaction Standard Exception

Any operation can raise the **InvalidTransaction** exception to indicate that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

7.2.5.2 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 a 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 a 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.)

WrongTransaction Exception

The **CosTransactions** module defines an exception that can be raised by the ORB when returning the response to a deferred synchronous request:

```
exception WrongTransaction {};
```

This exception can be raised only if the request is implicitly associated with a transaction (the current transaction at the time the request was issued).

The `get_response()` operation (defined on the **Request** interface) may raise the **WrongTransaction** exception if the transaction associated with the request is not the same as the transaction associated with the thread invoking `get_response()`.

The `get_next_response()` operation (defined on the **Orb** interface) may raise the **WrongTransaction** exception if the thread invoking `get_next_response()` has a non-null current transaction that is different than the one associated with the request.

7.2.5.3 Other Exceptions

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

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

These exceptions are described below along with the operations that raise them.

7.3 Transaction Service Interfaces

The interfaces defined by the Transaction Service reside in the **CosTransaction** module. (OMG IDL for the **CosTransactions** module is shown in Section 7.6 on page 202.) The interfaces for the Transaction Service are as follows:

- **Current**
- **TransactionFactory**
- **Terminator**
- **Coordinator**
- **RecoveryCoordinator**
- **Resource**
- **SubtransactionAwareResource**
- **TransactionalObject.**

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.

7.3.1 The 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 behaviour depends upon and may alter the transaction context associated with the invoking thread.

```
interface 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);
};
```

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.

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 7.3.4 on page 164 and Section 7.3.5 on page 165 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.

rollback()

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

Otherwise, the transaction associated with the client thread is rolled back. The effect of this request is equivalent to performing the **rollback()** operation on the corresponding terminator object (see Section 7.3.4 on page 164).

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.

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 7.3.5 on page 165).

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 7.3.5 on page 165).

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 7.3.5 on page 165).

set_timeout()

This operation modifies a state variable associated with the target object that affects the timeout period associated with top-level transactions created by subsequent invocations of the **begin()** operation. If the parameter has a non-zero value **n**, then top-level transactions created by subsequent invocations of **begin()** will be subject to being rolled back if they do not complete before **n** seconds after their creation. If the parameter is zero, then no application specified timeout is established.

get_control()

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

suspend()

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

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 7.3.3 on page 163 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 **TransactionRolledBack** exception.

7.3.2 The TransactionFactory Interface

The **TransactionFactory** interface is provided to allow the transaction originator to begin a transaction. This interface defines a single operation, **create()**, which creates a new top-level transaction.

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

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 non-zero 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 timeout is established.

7.3.3 The 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 new transaction is obtained using the **create()** operation 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 7.3.1 on page 160). (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 object to be transmitted to or used in other execution environments; at a minimum, it can be used within a single thread.

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.

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.

7.3.4 The 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.

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 **TransactionRolledBack** 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 7.2.5 on page 158). 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 transactional 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.

rollback()

The transaction is rolled back.

When a transaction is rolled back, all changes to transactional 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.

7.3.5 The 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_subtran_aware(in SubtransactionAwareResource r)
        raises(Inactive, NotSubtransaction);

    void rollback_only()
        raises(Inactive);

    string get_transaction_name();

    Control create_subtransaction()
        raises(SubtransactionsUnavailable, Inactive);
};
```

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.

get_status()

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

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.

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.

is_same_transaction()

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

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**.

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).

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**.

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.

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.

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.

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 **TransactionRolledBack** 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.

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 **TransactionRolledBack** may be raised if the subtransaction (or any ancestor) has been marked rollback only.

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.

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.

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 an 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 object to be transmitted to or used in other execution environments.

7.3.6 The RecoveryCoordinator Interface

A recoverable object uses a **RecoveryCoordinator** to drive the recovery process in certain situations. Each object supporting the **RecoveryCoordinator** interface is implicitly associated with a single resource registration request (invoking the **register_resource()** operation) and may be used only by that resource.

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

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. The **NotPrepared** exception is raised if the resource has not been prepared. This operation returns the current status of the transaction.

7.3.7 The 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, a resource should be prepared receive duplicate requests for the **commit()** or **rollback()** operation and to respond consistently.

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

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.

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 7.2.5 on page 158) 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.

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 7.2.5 on page 158) 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.

commit_one_phase()

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

forget()

This operation is performed only if the resource raised a heuristic outcome exception to **rollback()** or **commit()**. The resource can forget all knowledge of the transaction.

7.3.8 The SubtransactionAwareResource Interface

Recoverable objects that implement nested transaction behaviour may support a specialisation of the **Resource** interface called the **SubtransactionAwareResource** interface. A recoverable object can be notified of the completion of a subtransaction by registering a specialised resource object that offers the **SubtransactionAwareResource** interface with the Transaction Service. This registration is done by using the **register()** or the **register_subtran_aware()** operation of the current **Coordinator** object. A recoverable object generally uses the **register()** 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()** 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();
};
```

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 **TransactionRolledBack**.

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 rolls back.

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.

7.3.9 The **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 propagated on requests to the object. If an object does not support the **TransactionalObject** interface, the ORB is not required to propagate the transaction context on requests to the object.

```
interface TransactionalObject {  
};
```

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

7.4 The User View

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

7.4.1 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 a pseudo-object called **Current**, provided by the Transaction Service, to associate the transaction context with the application thread of control. In direct context management, an application manipulates the object and the other objects associated with the transaction.

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** pseudo-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.

7.4.1.1 Direct Context Management

Implicit Propagation

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

Explicit Propagation

The client application directly accesses the 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.

7.4.1.2 *Indirect Context Management*

Implicit Propagation

The client application uses operations on the **Current** pseudo-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.

Explicit Propagation

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

7.4.2 Interfaces

Function	Used By	Context Management	
		Direct	Indirect*
Create a transaction	Transaction originator	Factory::create () Control::get_terminator () Control::get_coordinator ()	begin () , set_timeout ()
Terminate a transaction	Transactor 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

Table 7-1 Use of Transaction Service Functionality

Note: For clarity, subtransaction operations are not shown.

* All indirect context management operations are on the **Current** pseudo-object interface.

7.4.3 Checked Transaction Behaviour

Some Transaction Service implementations will enforce checked behaviour 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. 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.

7.4.4 X/Open Checked Transactions

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 Distributed TP Transaction Managers are examples of transaction management functions which implement checked transaction behaviour.

This implementation of checked behaviour 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 beginning 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, in general, cannot know which objects are currently involved in the transaction, or may be in the future; that is, a request tree cannot be constructed or assured. Therefore, the use of explicit propagation is not permitted by a Transaction Service implementation which enforces X/Open style checked behaviour.

Applications that use synchronous requests implicitly exhibit checked behaviour. 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.

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.

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.

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.

7.4.5 Implementing a Transactional Client: Heuristic Completions

commit() takes the boolean **report_heuristics** input. If the **report_heuristics** argument is **false**, the **commit()** operation can complete as soon as the **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 **Resource** objects 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.

7.4.6 Implementing a Recoverable Server

A recoverable server includes at least one transactional object and one resource object. The responsibilities of each of these objects are as follows.

Transactional Object

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

The **Resource** object identifies the involvement of the recoverable server 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 server is concurrently involved.

A transactional 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 transactional object to determine whether the transaction associated with the request is one in which the transactional object is already registered.

The **hash_transaction()** operation allow the transactional 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.

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 7.5.1 on page 185.

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 transactional object can register a **Resource** object that replies **VoteReadOnly** to **prepare()** if its integrity constraints are satisfied (for example, 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.

7.4.7 Application Portability

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

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.

X/Open Checked Transactions

Transaction Service implementations may implement checked or unchecked behaviour. 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 guarantees 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 behaviour.

7.4.8 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 implementor. There is no change in the function provided.

7.4.9 Applications Using Both Checked and Unchecked Services

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

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

1. 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.
2. An application, whose resources are managed by a checked Transaction Service, may act as a client of unchecked objects, and preserve its checked semantics.

7.4.10 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.

Transaction Originator: Indirect and Implicit

In the code fragments below, a transaction originator uses indirect context management and implicit transaction propagation; `txn_crt` is a pseudo-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);
...
```

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 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 object reference is passed as an explicit parameter of the request: it is declared in the interface OMG IDL.

```
...
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);
```

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 should check whether it has already been registered for the same transaction. This is done using the `hash_transaction()` and `is_same_transaction()` operations. Note that this object registers itself as a resource. This imposes the restriction that the object may only be involved in one transaction at a time.

If more parallelism is required, separate resource objects should be registered for involvement in the same transaction.

```

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

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

```

Example of a Transactional Object

A 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

```

7.4.11 Model Interoperability

The Transaction Service supports interoperability between Transaction Service applications using implicit context propagation and procedural applications using the X/Open Distributed TP 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.

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.

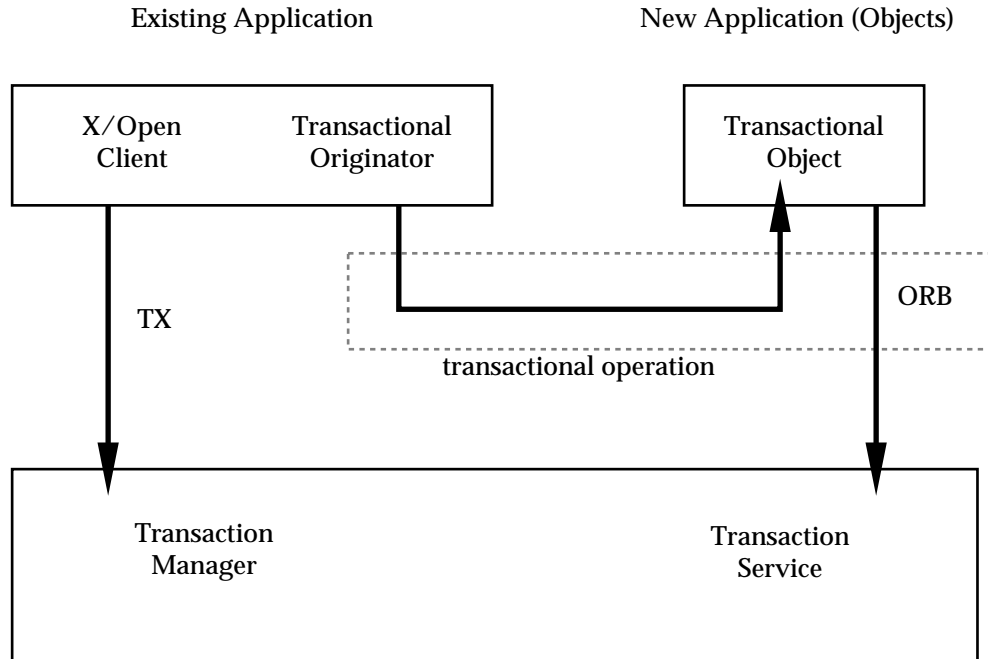


Figure 7-3 X/Open Client

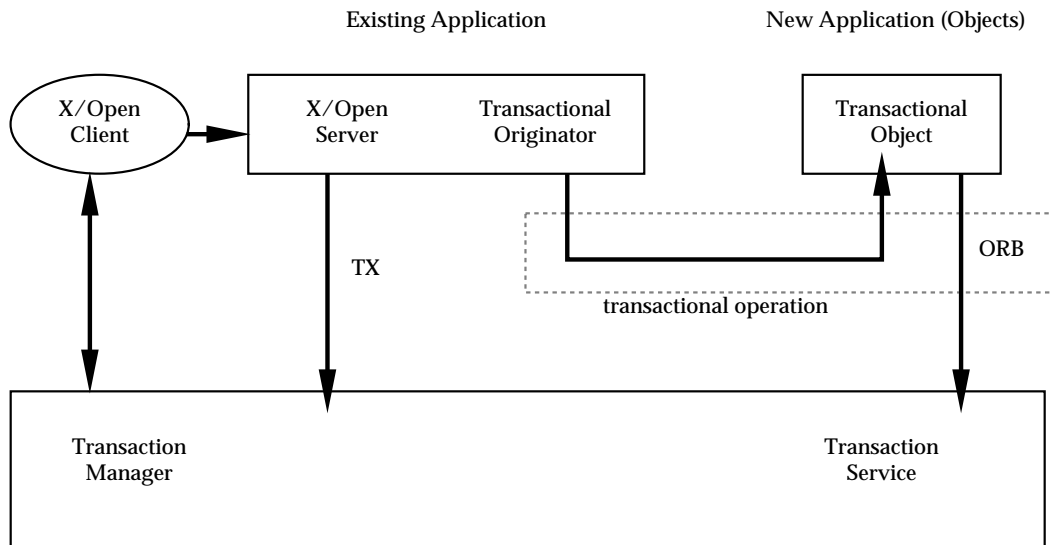


Figure 7-4 X/Open Server

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.

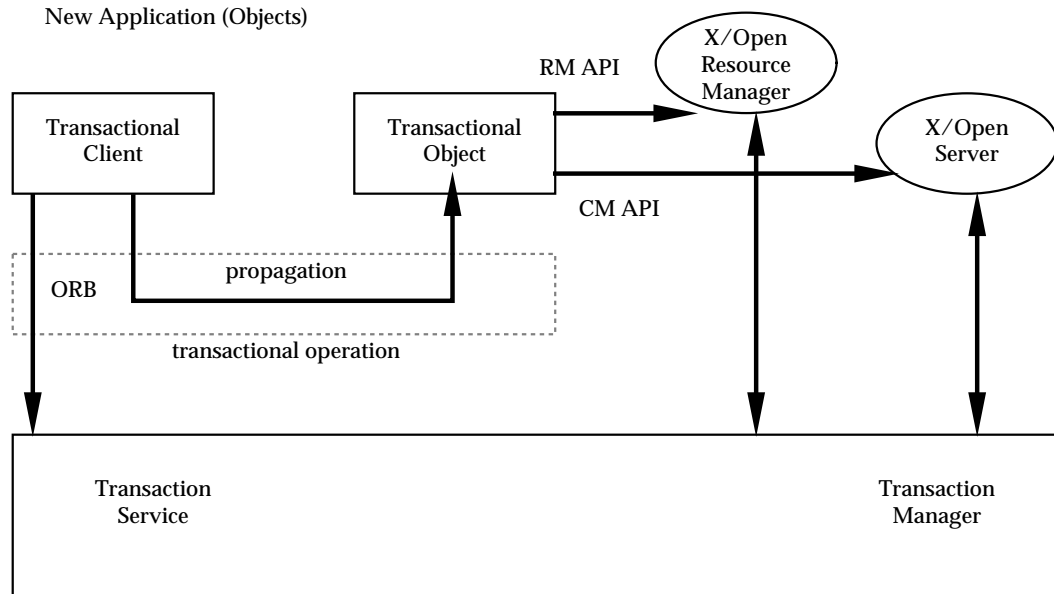


Figure 7-5 Example Transaction Export

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. How the X/Open transaction is associated with the Transaction Services transaction context is implementation-dependent.

A transactional object with a **Current** pseudo-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** pseudo-object is implementation-dependent.

7.4.12 Failure Models

The Transaction Service provides atomic outcomes for transactions in the presence of application, system or communication failures. This section describes the behaviour of application entities when failures occur. The protocols used to achieve this behaviour are described in Section 7.5.1 on page 185.

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); for example, failure of another object or failure in the communication with that object.

7.4.12.1 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 `TransactionRolledBack` 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 but this is not reliable since the status `NoTransaction` is ambiguous; it can 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 colocated in the same failure domain (for example, the same execution environment).

7.4.12.2 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 rolls 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 **TransactionRolledBack** exception will be returned to a client issuing **commit()**.

7.4.12.3 Recoverable Server

Behaviour 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. This protocol, including the local and external failure models and the required behaviour of the **Resource**, is described in Section 7.5.1 on page 185.

7.5 The Implementor View

This section contains three major categories of information:

- Section 7.5.1 defines in more detail the protocols of the Transaction Service for ensuring atomicity of transactions, even in the presence of failure.

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

- Section 7.5.2 on page 194 provides additional information for implementors of ORBs and Transaction Services in those areas where cooperation between the two is necessary to realise the Transaction Service function.

The following aspects of ORB and Transaction Service implementation are covered:

- transaction propagation
 - interoperation between different transaction service implementations
 - ORB changes necessary to support portability of transaction service implementations.
- Section 7.5.3 on page 200 describes how an implementation achieves interoperation between the Transaction Service and procedural Transaction Managers.

7.5.1 Transaction Service Protocols

The Transaction Service requires that certain protocols are followed to implement the atomicity property. These protocols affect the implementation of recoverable servers; that is, recoverable objects that register for participation in the two-phase commit process, and the **Coordinators** which are created by a transaction factory. These responsibilities ensure the proper execution of the two-phase commit protocol and include maintaining state information in stable storage, so that transactions can be properly completed even in the event of failure.

7.5.1.1 General Principles

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

1. The protocol defined by this specification is a two-phase commit with presumed abort.
This permits efficient implementations to be realised since the root **Coordinator** does not need to log anything before the commit decision, and the participants (that is, **Resource** objects) do not need to log anything before they prepare.
2. **Resource** objects — including subordinate **Coordinators** — do not start commitment by themselves, but wait for **prepare()** to be invoked.
3. **prepare()** is issued at most once to each **Resource**.
4. Participants must remember heuristic decisions until the **Coordinator** or some management application instructs them to **forget()** that decision.

5. A **Coordinator** knows which **Resource** objects are registered in a transaction and so is aware of **Resources** that have completed commitment.

In general, the **Coordinator** must remember this information if a transaction commits in order to ensure proper completion of the transaction. **Resources** can be forgotten early if they do not vote to commit the transaction.

6. A participant should be able to request the outcome of a transaction at any time, including after failures occurring subsequent to its **Resource** object being prepared.
7. Participants should be able to report the completion of the transaction (including any heuristic condition).

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

7.5.1.2 Normal Transaction Completion

Transaction completion can occur in two ways: as part of the normal execution of the **Current::commit** or **Terminator::commit** operations or independent of these operations if a failure should occur before normal execution can complete. This section describes the normal (that is, no failure) case. Section 7.5.1 on page 185 describes the failure cases.

Coordinator Role

The root **Coordinator** implements the following protocol:

- When the client asks to **commit()** the transaction, and no prior attempt to rollback the transaction has been made, the **Coordinator** issues the **prepare()** request to all registered **Resources**.
- If all registered **Resources** reply **VoteReadOnly**, then the root **Coordinator** replies to the client that the transaction committed (assuming that the client can still be reached).
- There is no requirement for the **Coordinator** to log in this case.
- If any registered **Resource** replies **VoteRollback** or cannot be reached, then the **Coordinator** will decide to **rollback()** and will so inform those registered **Resources** which already replied **VoteCommit**.

Once a **VoteRollback** reply is received, a **Coordinator** need not send **prepare()** to the remaining **Resources**. **rollback()** will be subsequently sent to **Resources** that replied **VoteCommit**. If the **report_heuristics** parameter was specified on **commit()**, the client will be informed of the **rollback()** outcome when any heuristic reports have been collected (and logged if required).

- Once at least one registered **Resource** has replied **VoteCommit** and all others have replied **VoteCommit** or **VoteReadOnly**, a root **Coordinator** may decide to commit the transaction.
- Before issuing **commit()** operations on those registered **Resource** objects which replied **VoteCommit**, the **Coordinator** must ensure that the commit decision and the list of registered **Resources** — those that replied **VoteCommit** — are stored in stable storage.
- If the **Coordinator** receives **VoteCommit** or **VoteReadOnly** responses from each registered **Resource**, it issues the **commit()** request to each registered **Resource** that responded **VoteCommit**.
- The root **Coordinator** issues **forget()** to a **Resource** after it receives a heuristic exception.

- This responsibility is not affected by failure of the **Coordinator**. When receiving **commit()** replies containing heuristic information, a **Coordinator** constructs a composite for the transaction.
- After having received all **commit()** operation replies, a root **Coordinator** forgets the transaction after having logged its heuristic status if heuristics reporting was requested by the originator.

The root **Coordinator** can now trigger the sending of the reply to the **commit()** operation if heuristic reporting is required. If no heuristic outcomes were recorded, the **Coordinator** can be destroyed.

One-phase Commit

If a **Coordinator** has only a single registered **Resource**, it can perform the **commit_one_phase()** operation on the **Resource** instead of performing **prepare()** and then commit or rollback. If a failure occurs, the **Coordinator** will not be informed of the transaction outcome.

Subtransactions

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

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

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

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

Recoverable Server Role

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

Top-level Registration

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

A recoverable object uses the `is_same_transaction()` operation to determine whether it is already registered in this transaction. It can also use `hash_transaction()` to reduce the number of comparisons — this relies on the definition of the `hash_transaction()` operation to return the same value for all **Coordinators** in the same transaction.

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

Subtransaction Registration

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

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

Top-level Completion

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

1. Returning **VoteCommit** to `prepare()`.
2. Before a **Resource** object replies **VoteCommit** to a `prepare()` operation, it must implement the following:
 - Make persistent the recoverable state of its recoverable object.
The method by which this is accomplished is implementation-dependent. If a recoverable object has only transient state, it need not be made persistent.
 - Ensure that its object reference is recorded in stable storage to allow it to participate in recovery in the event of failure.
How object references are made persistent and then regenerated after a failure is outside the scope of this specification. The Persistent Object Service or some other mechanism may be used. How persistent **Resource** objects get restarted after a failure is also outside the scope of this specification.
 - Record the **RecoveryCoordinator** object reference so that it can initiate recovery of the transaction later if necessary.
 - The **Resource** then waits for the **Coordinator** to invoke `commit()` or `rollback()`.
 - A **Resource** with a heuristic outcome must not discard that information until it receives a forget from its **Coordinator** or some administrative component.
3. Returning **VoteRollback** to `prepare()`.

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

4. Returning **VoteReadOnly** to **prepare()**.

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

Subtransaction Completion

The role of the **SubtransactionAwareResource** at subtransaction completion is defined by the **SubtransactionAwareResource** itself. The **Coordinator** only requires that it respond to **commit_subtransaction()** or **rollback_subtransaction()**.

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

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

Using a concurrency service as an example, such a resource in a nested transaction might want to know when the subtransaction commits because some other subtransaction may be waiting for a lock held by that subtransaction. Once that subtransaction commits, others may be granted the lock. There is no requirement to make lock ownership persistent until a **prepare()** message is received.

As for a persistence service, there are issues with keeping separate update information associated with a subtransaction. When that subtransaction commits, the persistence service may need to reorganise its information (such as undo information) in case the parent subtransaction chooses to **rollback()**. But again, the Persistent Object Service **Resource** need not be concerned with making updates permanent until a **prepare()** message is received. At that point, it has the same responsibilities as a top-level resource.

Subordinate Coordinator Role

An implementation of the Transaction Service may interpose subordinate **Coordinators** to optimise the commit tree for completing the transaction. Such **Coordinators** behave as transaction participants to their superiors and as **Coordinators** to their **Resources** or inferior **Coordinators**.

Registration

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

Subtransaction Registration

If any of the **Resources** registered with the subordinate **Coordinator** support the **SubtransactionAwareResource** interface, the subordinate **Coordinator** must register a **SubtransactionAwareResource** with its parent **Coordinator**. If any of the **Resources** registered with the subordinate using the **register_resource()** operation, the subordinate must register a **Resource** with its superior. If both types of resource were registered with the subordinate, the subordinate only needs to register a **SubtransactionAwareResource** with its superior.

Top-level Completion

A subordinate **Coordinator** implements the completion behaviour of a recoverable server.

Subtransaction Completion

A subordinate **Coordinator** implements the subtransaction completion behaviour of a recoverable server.

Subordinate Coordinator

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

- A subordinate **Coordinator** issues the **prepare()** operation to its registered **Resources** when it receives a **prepare()** request from its superior.

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

- If all registered **Resources** reply **VoteReadOnly**, then the subordinate **Coordinator** replies **VoteReadOnly** to its superior.

There is no requirement for the subordinate **Coordinator** to log in this case; the subordinate **Coordinator** takes no further part in the transaction and can be destroyed.

- If any registered **Resource** replies **VoteRollback** or cannot be reached then the subordinate **Coordinator** will decide to rollback and will so inform those registered **Resources** which already replied **VoteCommit**.

Once a **VoteRollback** reply is received, the subordinate **Coordinator** need not send **prepare()** to the remaining **Resources**. The subordinate **Coordinator** will reply **VoteRollback** to its superior.

- Once at least one registered **Resource** has replied **VoteCommit** and all others have replied **VoteCommit** or **VoteReadOnly**, a subordinate **Coordinator** may decide to reply **VoteCommit**.
- A subordinate **Coordinator** issues the **commit()** operation to its registered **Resources** which replied **VoteCommit** when it receives a **commit()** request from its superior.

If any **Resource** reports a heuristic outcome, the subordinate **Coordinator** reports a heuristic outcome to its superior. The specific outcome reported depends on the other heuristic outcomes received. The subordinate **Coordinator** should record the heuristic outcome in stable storage.

- After having received all commit replies, a subordinate **Coordinator** logs its heuristic status (if any).
- The subordinate **Coordinator** issues a **commit()** reply to its superior **Coordinator**.

If no heuristic report was sent, the **Coordinator** is destroyed.

- A subordinate **Coordinator** performs the **rollback()** operation on its registered **Resources** when it receives a **rollback()** request from its superior.

If any **Resource** reports a heuristic outcome, the subordinate **Coordinator** records the appropriate heuristic outcome in stable storage and reports this outcome to its superior.

- If a subordinate **Coordinator** receives a **commit_one_phase()** request, and it has a single registered **Resource**, it can simply perform the **commit_one_phase()** request on its **Resource**.
If it has multiple registered **Resources**, it behaves like a superior **Coordinator**, issuing **prepare()** to each **Resource** to determine the outcome, before issuing **commit()** or **rollback()** requests.
- A subordinate **Coordinator** performs the **forget()** operation on those registered **Resources** that reported a heuristic outcome when it receives a **forget()** request from its superior.

Subtransactions

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

7.5.1.3 Failure and Recovery

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

Failure Processing

The unit of failure is termed the *failure domain*. It may consist of the **Coordinator** and some local **Resource(s)** registered with it, or the **Coordinator** and the **Resources** may each be in their own failure domain.

Local Failure

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

A **Coordinator** is restarted only if it has logged the **commit()** decision.

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

If any registered objects no longer exist then this means that they completed commitment before the **Coordinator** failed and have no heuristic information.

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

The subordinate will inform its registered **Resources**.

External Failure

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

7.5.1.4 Transaction Completion after Failure

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

Resources

A **Resource** represents some collection of recoverable data associated with a transaction. It supports the **Resource** interface described in Section 7.3.7 on page 168. When recovering from failure after its changes have been prepared, a **Resource** uses the `replay_completion()` operation on the **RecoveryCoordinator** to determine the outcome of the transaction and continue completion.

Heuristic Reporting

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

Coordinator Role

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

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

Subtransactions

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

Recoverable Server Role

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

Once having replied **VoteCommit**, the **Resource** remains responsible for discovering the outcome of the transaction; that is, whether to commit or rollback. If the **Resource** subsequently makes a heuristic decision, this does not change its responsibilities to discover the outcome.

If No Heuristic Decision is Made

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

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

When a Heuristic Decision is Made

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

The heuristic outcome details must be retained persistently until the **Resource** is instructed to forget. Thus in this case the **Resource** remains persistent until the **forget()** is received.

Subordinate Coordinator Role

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

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

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

7.5.2 ORB/TS Implementation Considerations

The Transaction Service and the ORB must cooperate to realise certain Transaction Service functions. This is discussed in greater detail in the following sections.

7.5.2.1 Transaction Propagation

The transaction is represented to the application by the object. Within the Transaction Service, an implicit context is maintained for all threads associated with a transaction. Although there is some common information, the implicit context is not the same as the object defined in this specification and is distinct from the ORB Context defined in CORBA 1.2. It is the implicit context that must be transferred between execution environments to support transaction propagation. The implicit context does not have an OMG IDL interface.

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

No OMG IDL declaration is required to cause propagation of the implicit context with a request. The minimum amount of information that could serve as a implicit context is the object reference of the **Coordinator**. However, an identifier (for example, an X/Open XID) is also required to allow efficient (local) execution of the `is_same_transaction()` and `hash_transaction()` operations when interposition is done. Implementations may choose to also include the **Terminator** object reference if they support the ability for ending the transaction in execution environments other than the originator's. Transferring the implicit context requires interaction between the Transaction Service and the ORB to add or extract the implicit context from ORB messages. This interaction is also used to implement the checking functions described in Section 7.4.4 on page 175.

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

Interposition

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

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

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

Many implementations of the Transaction Service will want to perform interposition and thus create **Control** objects and subsequently **Coordinator** objects for each execution environment participating in the transaction. To create a new (local) **Control**, an importing Transaction

Service uses the information in the implicit context and some local factory. Interposition must be complete before the `get_control()` operation can complete in the target object. An object adaptor is one possible place to implement interposition.

Subordinate Coordinator Registration

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

7.5.2.2 Transaction Service Interoperation

The Transaction Service can be implemented by multiple components at different locations. The different components can be based on the same or different implementations of the Transaction Service. As stated in Section 7.1.5 on page 151, it is a requirement that multiple Transaction Services interoperate across the same ORB and different ORBs. Transaction Service interoperation across different ORBs cannot be specified in the absence of ORB interoperability.

Transaction Service interoperation across a single ORB is specified by defining the data structures exported between different implementations of the Transaction Service. These data structures are of two types:

- structures which are defined by the operations of the Transaction Service and their associated OMG IDL — these structures are specified completely in Section 7.3 on page 160
- structures which are specific to the boundary between the ORB and the Transaction Service — this structure applies only when the implicit context is exported to a different Transaction Service domain.

When the implicit context is propagated with a request, the destination uses it to locate the superior **Coordinator**. That **Coordinator** may be implemented by a foreign Transaction Service. By registering a **Resource** with that **Coordinator**, the destination arranges to receive two-phase commit requests from the (possibly foreign) Transaction Service.

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

An interface between the ORB and the Transaction Service is defined that arranges for the implicit context to be carried on messages that represent method invocations on transactional objects. This interface is described in Section 7.5.2 on page 194.

Structure of the PropagationContext

The **PropagationContext** structure is defined by the following OMG IDL:

```

module CosTSInteroperation { // PIDL
    struct otid_t {
        long formatID; /*format identifier. 0 is OSI TP */
        long bequal_length;
        sequence <octet> tid;
    };
    struct TransIdentity {
        CosTransactions::Coordinator coordinator;
        CosTransactions::Terminator terminator;
        otid_t otid;
    };
    struct PropagationContext {
        unsigned long timeout;
        TransIdentity current;
        sequence <TransIdentity> parents;
        any implementation_specific_data;
    };
};

```

For the functions defined within the base section of the propagation context, it is necessary only to send it with requests. Implementations may use the vendor-specific portion for additional functions (for example, to register an interposed **Coordinator** with its superior) which may require the propagation context to be returned. Whether it is returned or not is implementation-specific.

otid_t

The **otid_t** structure is a more efficient OMG IDL version of the X/Open-defined transaction identifier.

TransIdentity

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

Coordinator

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

Terminator

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

otid

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

timeout

The timeout value associated with the transaction in the relevant `set_timeout()` operation (or the default timeout).

<TransIdentity> Parents

A sequence of **TransIdentity** structures representing the parent(s) of the current transaction. The ordering of the sequence starts at the parent of the current transaction and includes all ancestors up to the top-level transaction. An implementation that does not support nested transactions would send an empty sequence. This allows a non-nested transaction implementation to know when a nested transaction is being imported. It also supports efficient (local) execution of the **Coordinator** operations which test parentage when the importing Transaction Service does interposition.

implementation_specific_data

This information is exported from an implementation and is required to be passed back with the rest of the context if the transaction is re-imported into that implementation.

Appearance of the Propagation Context in Messages

To specify how the propagation context appears in messages, it is regarded as an extra, implicit argument which is effectively added to the signatures of transactional operations. The specification simply describes how the original operation signature is transformed with the new argument.

A transactional ORB supporting the target object will receive a request with a signature defined as:

```
result_type op( arg1,..., argN);
```

but will actually receive, and must reply, as though the signature were:

```
result_type op(
    arg1,..., argN,
    inout CostSInteroperation::PropagationContext ctx);
```

7.5.2.3 Transaction Service Portability

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

Because it is recognised that other object services and future extensions to the CORBA Specification may require similar mechanisms (for example, the Security Service may need to pass authentication information with requests), this component is specified separately from the main body of the Transaction Service to allow it to be revised or replaced by a mechanism common to several services independently of any future Transaction Service revisions.

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

1. An additional ORB interface that allows the Transaction Service to identify itself to the ORB when present in order to be involved in the transmission of transactional requests.

2. A collection of OTS operations (the OTS callbacks) that the ORB invokes when a transactional request is sent and received.

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

Identification of the Transaction Service to the ORB

Prior to the first transactional request, the Transaction Service will identify itself to the ORB within its domain to establish the transaction callbacks to be used for transactional requests and replies. This is accomplished using the following interface:

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

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

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

NotAvailable

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

AlreadyIdentified

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

identify_sender()

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

identify_receiver()

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

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

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

7.5.2.4 The Transaction Service Callbacks

The CosTSPortability module defines two interfaces. Both interfaces are defined as PIDL. The sender interface defines a pair of operations which are called by the ORB sending the request before it is sent and after its reply is received. The receiver interface defines a pair of operations which are called by the ORB receiving the request when the request is received and before its reply is sent. Both interfaces use the PropagationContext structure defined in Section 7.5.2 on page 194.

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

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

ReqId

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

Sender::sending_request

A request is about to be sent. The Transaction Service returns a propagation context to be delivered to the Transaction Service at the server managing the target object. The TransactionRequired standard exception is raised when invoked outside the scope of a transaction.

Sender::received_reply

A reply has been received. The propagation context from the server is passed to the Transaction Service along with the returned environment. The Transaction Service examines the environment to determine whether the request was successfully performed. If the environment indicates the request was unsuccessful, the TransactionRolledBack standard exception is raised.

Receiver::received_request

A request has been received. The propagation context defines the transaction making the request.

Receiver::sending_reply

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

7.5.2.5 Behaviour of the Callback Interfaces

The following sections describe the protocols associated with the callback interfaces.

Requirements on the ORB

The ORB will invoke the sender callbacks only when a transactional operation is issued for an object in a different process. Objects within the same process implicitly share the same transaction context. The receiver callbacks are invoked when the ORB receives a transactional request from a different process.

The ORB must generate a request identifier for each outgoing request and be able to associate the identifier with the reply when it is returned. For deferred synchronous invocations, this allows the Transaction Service to correlate the reply with the request to implement checked behaviour. The request identifier is passed on synchronous invocations to permit the same interface specification to be used.

The callbacks are invoked in line with the processing of requests and replies. This means that the callbacks will be executed on the same thread that issued or processed the actual request or reply. When the dynamic invocation interface (DII) is used, the `received_reply()` callback must be invoked on the same thread that will subsequently process the response.

Requirements on the Transaction Service

Within a single process, the transaction context is part of the thread-specific state. Multiple threads executing on behalf of the same transaction will share the same transaction context since a thread can only execute on behalf of a single transaction at a time. Since the callbacks are defined as PIDL (that is, procedure calls), they are invoked on the client's thread when sending and the server's thread when receiving. This enables the Transaction Service to locate the proper transaction context when sending and associate the received transaction context with the thread that will process the transactional operation. The callback interfaces may only raise standard exceptions and may not make additional object invocations using the ORB.

7.5.3 Model Interoperability

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

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

This means that a single transaction management service can provide the Transaction and Transaction Manager interfaces of the Transaction Service, and also provide the TX and XA interfaces of X/Open Distributed TP. This is illustrated in Figure 7-6 on page 201.

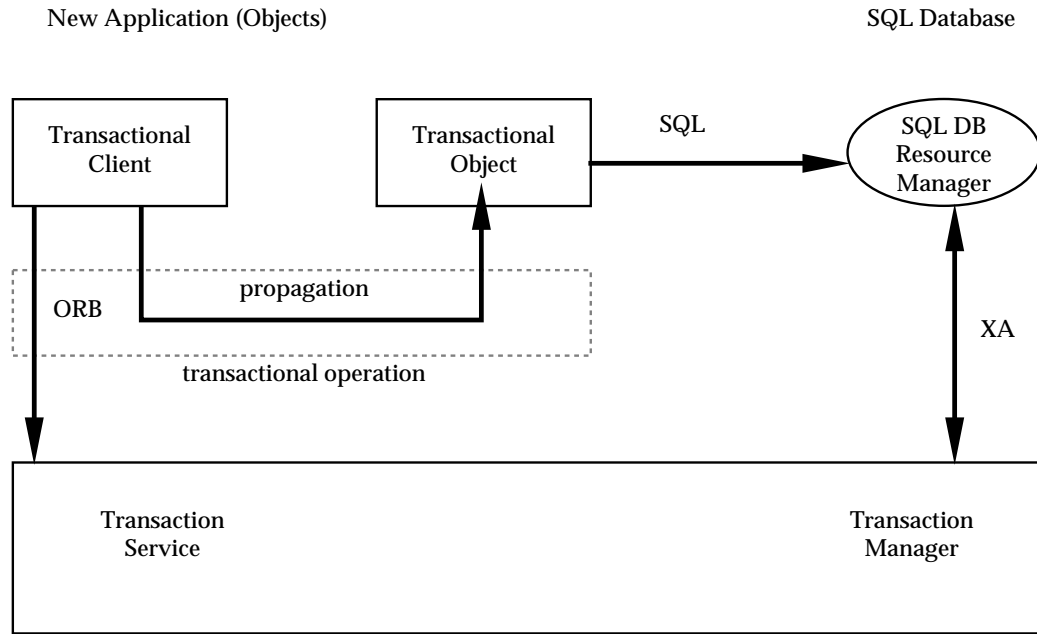


Figure 7-6 Model Interoperability Example

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

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

7.6 The CosTransactions Module

```

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

enum Vote {
    VoteCommit,
    VoteRollback,
    VoteReadOnly };

// Standard exceptions
exception TransactionRequired {};
exception TransactionRolledBack {};
exception InvalidTransaction {};

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

// Exception from Orb operations
exception WrongTransaction {};

// Other transaction-specific exceptions
exception SubtransactionsUnavailable {};
exception NotSubtransaction {};
exception Inactive {};
exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};

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

// Current transaction pseudo object (PIDL)
interface 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);
};

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

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

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

interface Coordinator {

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

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

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

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

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

```

```

        raises(Inactive);

    string get_transaction_name();

    Control create_subtransaction()
        raises(SubtransactionsUnavailable, Inactive);
};

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

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

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

interface TransactionalObject {
};
}; // End of CosTransactions Module

```

7.6.1 The CosTSInteroperation Module

```

module CosTSInteroperation { // PIDL
    struct otid_tid {
        long formatID; /*format identifier. 0 is OSI TP */
        long bequal_length;
        sequence <octet> tid;
    };
    struct TransIdentity {
        CosTransactions::Coordinator coordinator;
        CosTransactions::Terminator terminator;
        otid_t otid;
    };
};

```

```
    struct PropagationContext {
        unsigned long timeout;
        TransIdentity current;
        sequence <TransIdentity> parents;
        any implementation_specific_data;
    };
};
```

7.6.2 The CosTSPortability Module

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

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


The Transaction Service and TP Standards

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

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

A.1 Support of X/Open TX interface

A.1.1 Requirements

The X/Open Distributed TP Model is now widely known and implemented.

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

A.1.2 TX Mappings

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

TX Interface	Current Interface
<i>tx_open()</i> <i>tx_close()</i>	no equivalent no equivalent
<i>tx_begin()</i> <i>tx_rollback()</i>	Current::begin() Current::rollback() or Current::rollback_only()
<i>tx_commit()</i> <i>tx_set_commit_return()</i> <i>tx_set_transaction_control()</i> <i>tx_set_transaction_timeout()</i> <i>tx_info()</i> — XID <i>tx_info()</i> — COMMIT_RETURN	Current::commit() report_heuristics parameter of Current::commit() no equivalent (chained transactions not supported) Current::set_timeout() Current::get_name()* no equivalent
<i>tx_info()</i> — TRANSACTION_TIME_OUT <i>tx_info()</i>	no equivalent Current::get_status()

tx_open()

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

This processing is also related to a future Initialization Service.

tx_close()

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

tx_begin()

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

tx_rollback()

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

tx_commit() and tx_set_commit_return()

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

tx_set_transaction_control()

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

tx_set_transaction_timeout()

tx_set_transaction_timeout() corresponds to **Current::set_timeout()**.

tx_info()

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

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

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

A.2 Support of X/Open Resource Managers

A.2.1 Requirements

X/Open DTP-compliant Resource Managers, simply called X/Open Resource Managers or RMs, are Resource Managers that can be involved in a distributed transaction by allowing their two-phase commit protocol to be controlled via the X/Open XA Interface. Many RDBMS suppliers currently offer (or intend to offer) X/Open Resource Managers. Many OODBMSs intend also to support the XA Interface (some have already implemented it).

The Transaction Service must therefore be able to interact with X/Open Resource Managers. This section will illustrate how an X/Open Resource Manager may be used by a Transaction Service-compliant system.

The architecture of Transaction Service, based on the same concepts as the X/Open Distributed TP Model, allows mapping of Transaction Service operations to and from XA interactions.

A.2.2 XA Mappings

This section gives an overall view of a possible mapping between XA primitives offered by an X/Open Resource Manager (called RM hereafter) and the interfaces of the Transaction Service and their operations in the different phases of a transaction and during recovery.

The mappings are summarised in the following table:

XA Interface	Object Transaction Service
<i>xa_start()</i>	Receiving Request
<i>ax_reg()</i>	Current::resume
<i>xa_end()</i>	Sending Reply
<i>ax_unreg()</i>	no equivalent
<i>xa_prepare()</i>	Resource::prepare
<i>xa_commit()</i>	Resource::commit
<i>xa_rollback()</i>	Resource::rollback
<i>xa_recover()</i>	no equivalent
no equivalent	RecoveryCoordinator::replay_completion ()
<i>xa_forget()</i>	Resource::forget

In the X/Open Distributed TP Model all the interactions are made in the same thread of control.

A.2.3 XID

An XID is the Transaction Identifier. As defined in the X/Open XA Specification, this XID is the only information used by Resource Managers to associate logged information to the transaction, including objects before images, after images, locks and transaction state.

The content of an XID is defined by X/Open as follows:

```
#define XIDDATASIZE 128 /* size in bytes */
#define MAXGTRIDSIZE 64
    /* maximum size in bytes of gtrid */
#define MAXBQUALSIZE 64
    /* maximum size in bytes of bqual */
```

```

struct xid_t {
    long formatID; /* format identifier */
    long gtrid_length;
        /* value not to exceed 64 */
    long bqual_length;
        /* value not to exceed 64 */
    char data [XIDDATASIZE];
};
typedef struct xid_t XID;

```

The XID uniquely and unambiguously identifies a distributed transaction (information contained in the “gtrid” part of the XID) and a transaction-branch, the work performed by a node in the transaction tree (information contained in the *bqual* part of the XID).

To facilitate the use of distributed transactions in heterogeneous environments, X/Open has adopted the structure of the Transaction Identifier used in OSI TP but it is allowed to use other Transaction Identifier formats, which may be defined by the value of a Format Identifier field contained in the XID structure. The OSI TP Transaction Identifier contains information about the initiator of the transaction and the superior in the transaction tree; this information may be used, during recovery, to contact these entities and obtain the outcome of the transaction.

In the Transaction Service, tightly-coupled concurrency is assumed (a lock held by a transaction may be accessed by any participant of the same transaction) and the transaction branch part of the XID must not be given to RMs.

A.2.3.1 Interactions with an XA-compliant RM

Model

To model the relationship between the XA interface and the Transaction Service operation, an X/Open Transaction Manager has been modelled; this component is used here as a way to describe the interactions and may be implemented in a different manner.

Propagation of a Transaction to an RM

An RM may support two kinds of involvement interactions:

- Static registration, in which the Transaction Service involves the RM whenever it is itself involved in a new transaction.
- Dynamic registration, in which the RM notifies the Transaction Service that it has been requested to perform some work and request the XID of the current transaction.

An RM gets involved in a transaction when it has to perform some new work for this transaction. This happens in one of the following situations:

- A request carrying a transaction context has just been received and the RM has to perform work for the target object of this request.
- A method performing a request that is carrying a transaction context is resumed (by a **Current::resume()** operation).

An object may receive several requests carrying a transaction context for the same transaction. An RM may also perform work for several objects in the same transaction. Thus, an RM may be involved several times in the same transaction; the “resume” and the “join” concepts of XA may be used to notify the RM of any multiple involvement. When an RM has to get involved in a transaction, it must obtain the corresponding XID from the Transaction Service through an

xa_start() primitive or by a return parameter of an *ax_reg()* primitive. This XID is transmitted to the RM as a parameter to *xa_start()* or *ax_reg()* and is used by the RM to relate any work performed or any lock obtained to the transaction.

If the Transaction Service is called by an *ax_reg()* while it is not aware of any transaction, it returns a null XID to the RM. The RM is then free to start a local transaction of its own, and no Transaction Service transaction will be accepted until the RM issues an *ax_unreg()*.

Refer to the X/Open XA Specification for more information about propagation of a transaction to an RM.

First Phase of Commitment

When the first phase of commitment is started, the Transaction Service issues an *xa_prepare()* primitive and process its results to determine its decision.

Second Phase of Commitment

When the second phase of commitment is started, the Transaction Service issues an *xa_commit()* primitive and process its results to determine the heuristic situation.

One-phase Commitment

When the Transaction Service wants to perform a one-phase commitment, it issues an *xa_commit()* primitive and processes its results to determine the heuristic situation.

In the XA interface, there is no specific primitive for one-phase commitment: an RM must consider an *xa_commit()* without preceding *xa_prepare()* as a request to perform a one-phase commitment.

Rollback

When a rollback has to be performed, the Transaction Service issues an *xa_rollback()* primitive and processes its results to determine the heuristic situation.

Recovery

In the XA interface, the recovery of an RM is triggered by the Transaction Manager which issues an *xa_recover()*; the RM then gives back a list of all XIDs that are either in the ready state or have been heuristically completed.

In the Transaction Service recovery is performed by a **Resource** that issues a **replay_completion()** operation to a transaction **Coordinator**.

Failure of an Operation

Any failure of an operation typically leads to a rollback of the transaction, especially if it is not possible to determine whether the operation has been performed or not. However, in the decided commit state, the commit operation must be retried until the reply has been received (unless a heuristic hazard condition is detected).

Failure of an RM

If an RM fails, the Transaction Service detecting the failure will issue an *xa_recover()*. The Transaction Service will then get a list of XIDs of transactions for which the RM is in the ready state and transactions that have been heuristically completed.

The Transaction Service will then:

- Call *xa_rollback()* for all transactions that it knows to be neither in the prepared state nor in the decided commit state.
- Call *xa_commit()* for all transactions that it knows to be in the decided commit state.
- Wait for the decisions commit or rollback for the other.

Failure of Transaction Service

Upon warm restart of the Transaction Service and retrieval of the states of transactions needing recovery from stable storage, the Transaction Service will call *xa_recover()* to get the list of transactions for which the RM needs recovery (see **Failure of an RM**).

A.3 Interoperation with Transactional Protocols

Transactional Protocols

A CORBA application may sometimes need to interoperate with one or more applications using one of the *de facto* standard transactional protocols: OSI TP and SNA LU 6.2. In this case, the Transaction Service must be able to import or export transactions using one of these protocols.

Export is the ability to relate a transaction of the Transaction Service to a transaction of a foreign transactional protocol. Importing means relating a Transaction Service transaction to a transaction started on a remote application and propagated via the foreign transactional protocol.

Since the model used by the Transaction Service is similar to the model of OSI TP and the X/Open Distributed TP Model, the interactions with OSI TP are straightforward. Since OSI TP is a compatible superset of SNA LU 6.2, a mapping to SNA communications is easily accomplished.

To interoperate, a mapping should be defined for the two-phase commit, rollback and recovery mechanisms, and for the transaction identifiers.

Note that neither OSI TP nor SNA LU 6.2 supports nested transactions.

A.3.1 OSI TP Interoperability

OSI TP is the transactional protocol defined by ISO. It has been selected by X/Open to allow the distribution of transactions by one of the communication interfaces: remote procedure call (see the X/Open TxRPC Specification), client/server (see the X/Open XATMI Specification) or peer-to-peer (see the X/Open CPI-C Specification).

The Transaction Service supports only unchained transactions. The use of dialogues using the Chained Transactions functional unit is possible only if restrictive rules are defined. These rules are not described in this document.

OSI TP Transaction Identifiers

In OSI TP, loosely-coupled transactions are supported and every node of the transaction tree possesses a transaction branch identifier which is composed of the transaction identifier (or atomic action identifier) and a branch identifier (the branch identifier being null for the root node of the transaction tree). Both the transaction identifier and the branch identifier contain an AE-Title (Application Entity Title) and a suffix that make it unique within a certain scope.

The format of the standard X/Open XID is compatible with the OSI TP identifiers, the *gtrid* corresponding to the atomic action identifier and the *bqual* corresponding to the branch identifier.

Incoming OSI TP Communications (Imported Transactions)

The Transaction Service is a subordinate in an OSI TP transaction tree and interacts with its superior by regular PDUs as defined by the OSI TP protocol. The Transaction Service stores internally the transaction identifier received on the OSI TP dialogue.

The Transaction Service maps the OSI TP commitment, rollback and recovery procedures to the Transaction Service commitment procedure as follows:

- The Transaction Service, upon reception of an OSI TP *Prepare* message, will enter the first phase of commitment procedure.

- When it enters the prepared state for the transaction, the Transaction Service will trigger the sending of an OSI TP *Ready* message to its superior. (It may trigger a *Recover (Ready)* message when normal communications are broken with the superior.)
- The Transaction Service, upon reception of an OSI TP *Commit* message, enters the second phase of commitment procedure. (It may receive a *Recover (Commit)* when normal communications are broken with the superior.)
- The Transaction Service, upon reception of an OSI TP *Rollback* message (it may be a *Recover (Unknown)* when normal communications are broken with the superior or any other rollback-initiating condition) will enter its rollback procedure (unless a rollback is already in progress).
- The Transaction Service, upon reception of the last rollback reply, will trigger the sending of a *Rollback Response/ Confirm* message to its superior.

Outgoing OSI TP Communications (Exported Transactions)

The Transaction Service behaves as a superior in an OSI TP transaction tree and interacts with its subordinates by regular PDUs as defined by the OSI TP protocol.

The Transaction Service will map the OSI TP commitment procedure as follows:

- The Transaction Service, during the first phase of commitment procedure, will invoke an OSI TP *Prepare* message to all its subordinates.
- Upon reception of an OSI TP *Ready* message, the Transaction Service will process this message as a successful reply to a *prepare()*.
- The Transaction Service, upon entering the second phase of the commitment procedure, will send an OSI TP *Commit* message (it may be a *Recover (Commit)* when normal communications are broken with the subordinate) to all subordinates.
- The Transaction Service, upon reception of an OSI TP *Rollback* message (it may be any other rollback-initiating condition) will enter its rollback procedure (unless a rollback is already in progress).
- The Transaction Service, upon reception of the last *Rollback Response/ Confirm* message from its subordinates, will process this message as a reply to a *rollback()* operation and determine the heuristic situation.

A.3.2 SNA LU 6.2 Interoperability

SNA LU 6.2 ([SNA88a], [SNA88b]) is a transactional protocol defined by IBM. It is widely used for transaction distribution. The standard interface to access LU 6.2 communications is CPI-C (Common Programming Interface for Communications) defined by IBM in the context of SAA [CPIC93]. This has been evolved by the CPI-C Implementers' Workshop to become CPI-C level 2, a modern interface usable for LU 6.2 and OSI TP communications (see the X/Open CPI-C Specification).

LU 6.2 supports only chained transactions but, at a given node, a transaction is started only when resources have been involved in the transaction. LU 6.2 can be used for a portion of an unchained transaction tree if the LU 6.2 conversations are ended after each transaction by any node that has both LU 6.2 conversations and dialogues of an unchained transaction.

LU 6.2 Transaction Identifiers

SNA LU 6.2 also supports loosely-coupled transactions and uses a specific format for transaction identifiers: the Logical Unit of Work (LUWID) corresponds to the OSI Transaction Identifier. The LUWID is composed of:

- the Fully Qualified Logical Unit Name, which is composed of up to 17 bytes, and is unique in an SNA network or a set of interconnected SNA networks
- an instance number which is unique at the LU that creates the transaction
- the sequence number that is incremented whenever the transaction is committed.

The Conversation Correlator corresponds to the OSI TP Branch Identifier; it is a string of 1 to 8 bytes which are unique within the context of the LU having established the conversation, and is meaningful when combined with the Fully Qualified LU Name of this Logical Unit.

Incoming LU 6.2 Communications

The LU 6.2 two-phase commit protocol is different from the OSI TP protocol: the system sending a *Prepare* message has to perform logging and is responsible for recovery. LU 6.2 does also support features like last-agent optimisation, read-only, and allows any node in the transaction tree to request commitment.

The Transaction Service is a subordinate in an LU 6.2 transaction tree and interacts with its superior using SNA requests and responses as defined by the LU 6.2 protocol. The Transaction Service internally stores the LUWID corresponding to the incoming conversation.

The Transaction Service maps the LU 6.2 commitment, rollback and recovery procedures to the Transaction Service commitment procedure as follows:

- The Transaction Service, upon reception of an LU 6.2 *Prepare* message, will enter the first phase of commitment procedure.
- The Transaction Service, upon entering the prepared state for the transaction, the Transaction Service will trigger the sending of a *Request Commit* message to its superior.
- The Transaction Service, upon reception of an LU 6.2 *Committed* message (it may be a *Compare States (Committed)* when normal communications are broken with the superior) will enter the second phase of commitment procedure.
- The Transaction Service, upon leaving the decided commit state, will trigger the sending of a *Forget* message to its superior (it may be a *Reset* when normal communications are broken with the superior).

Due to the two-phase commit difference, the Transaction Service will never send the equivalent of the *Recover (Ready)* unless prompted by the superior.

The last-agent and read-only features may also be supported by the Transaction Service.

Outgoing LU 6.2 Communications

The Transaction Service has to log when the *Prepare* message is sent and, in case of communication failure or restart of the Transaction Service, a recovery is needed.

A.3.3 ODMG Standard

ODMG-93 is a standard defined by ODMG (Object Database Management Group) describing portable interface to access Object Database Management Systems (ODBMS).

Since it is likely that, in the future, a good percentage of objects involved in transactions will be handled by an ODBMS, this standard has a strong relationship with the Transaction Service.

A.4 ODMG Model

The ODMG model defines optional transactions and supports the nested transaction concept. The ODMG model does not cover the integration of ODBMS with an external transaction service, which allows other resources and communications to be involved in a transaction. No two-phase commit or recovery protocol is described.

A transaction object has to be created. The transactional operations are:

- *Begin* (or start) to begin a transaction (or a subtransaction).
- *Commit* to request commitment of a transaction.
- *Abort* to rollback a transaction.
- *Checkpoint* to commit the transaction but keep the locks. This feature is not supported by the current version of the Transaction Service.
- *abort_to_top_level* to request rollback of a nested transaction family. The Transaction Service does not directly support this feature but does provide means to perform this functionality by resuming the context of the top-level transaction and then requesting rollback.

If the transaction object is destroyed, the transaction is rolled back.

Integration of ODMG ODBMSs with the Transaction Service

Since ODMG-93 does not define any way to integrate an ODBMS into an existing transaction, the integration is difficult unless the ODBMS does support the XA interface, in which case Section A.2.3.1 on page 210 is applicable.

In the future, it is anticipated that ODBMS will implement the Transaction Service-defined interfaces and be considered as a recoverable server.

A possibility is to use, at a root node, an ODBMS as a last resource and, after all subordinates are prepared, to request a one-phase commitment to the ODBMS. If the outcome for the ODBMS is commit, the transaction will be committed, if it is rollback, the transaction will be rolled back. The mechanism may work if it is possible to determine, after a crash, whether the ODBMS committed or rolled back (this may be done at application level).

Glossary

2PC

See **Two-phase Commit** on page 224.

Abort

See **Rollback** on page 223.

Active

The state of a transaction when processing is in progress and completion of the transaction has not yet commenced.

Atomicity

A transaction property that ensures that if work is interrupted by failure, any partially completed results will be undone. A transaction whose work completes is said to *commit*. A transaction whose work is completely undone is said to *rollback (abort)*.

Begin

An operation on the Transaction Service which establishes the initial boundary of a transaction.

Bind

To bind a name is to create a name binding in a given context.

Commit

Commit has two definitions as follows:

1. An operation in the **Current** and **Terminator** interfaces that a program uses to request that the current transaction terminates normally and that the effects of that transaction be made permanent.
2. An operation in the **Resource** interface which causes the effects of a transaction to be made permanent.

Commit Coordinator

In a two-phase commit protocol, the program that collects the vote from the participants.

Commit Participant

In a two-phase commit protocol, the program that returns a vote on the completion of a transaction.

Committed

The property of a transaction or a transactional object, when it has successfully performed the commit protocol. See also **In-doubt** on page 221, **Active** and **Completion**.

Completion

The processing required (either by commit or abort) to obtain the durable outcome of a transaction.

Compound Name

A name with multiple components. A sequence of names that defines a path in the naming graph to navigate the resolution process.

Cooperation

An interface of the Transaction Service which allows it to track transactional operations and propagate transaction context with other transaction services in the current transaction. This is an optional interface that allows portability of the Transaction Service.

Coordinator

A **Coordinator** object involves **Resource** objects in a transaction when they are registered. A **Coordinator** is responsible for driving the two-phase commit protocol. See also **Commit Coordinator** on page 219 and **Commit Participant** on page 219.

Consistency

A property of a transaction that ensures that the transaction's actions, taken as a group, do not violate any of the integrity constraints associated with the state of its associated objects. This requires that the application program is implemented correctly. The Transaction Service provides the functionality to support application data consistency.

CORBA

Common Object Request Broker Architecture.

Decided Commit State

A root transaction **Coordinator** enters the decided commit state when it has written a log-commit record; a subordinate transaction **Coordinator** or **Resource** object is in the decided commit state when it has received the commit instruction from its superior; in the latter case, a log-commit record may be written, but this is not essential.

Decided Rollback State

A transaction **Coordinator** or **Resource** object enters the decided rollback state when it decides to rollback the transaction or has received a signal to do so.

Direct Context Management

An application manipulates the object and the other objects associated with the transaction. See also **Indirect Context Management** on page 221.

Durability

A transaction property that ensures the results of a successfully completed transaction will never be lost, except in the event of catastrophe. It is generally implemented by a combination of persistent storage and a logging service that provides a backup copy of permanent changes.

Event

A state change of an object that causes the behaviour of an object.

Event Channel

An intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

Execution Environment

An implementation-dependent factor that may determine the outcome of certain operations on the Transaction Service. Typically, the execution environment is the scope within which shared state is managed.

Factory Object

An object that creates another object.

Federation

The principle whereby each component retains its autonomy rather than becoming subordinate to another.

Flat Transaction

A transaction that has no subtransactions, and that cannot have subtransactions.

Forgotten State

This is not really a transaction state at all, because there is no memory of the transaction; it has either completed or rolled back and all records on permanent storage have been deleted.

Heuristic Commit or Abort

To unilaterally make the commit or abort decision about in-doubt transactions when the **Coordinator** fails or contact with the **Coordinator** fails.

Indirect Context Management

An application uses the **Current** pseudo-object, provided by the Transaction Service, to associate the transaction context with the application thread of control. See also **Direct Context Management** on page 220.

In-doubt

The state of a transaction if it is controlled by a Transaction Manager that cannot be contacted, so the commit decision is in doubt. See also **Active** on page 219 and **Committed** on page 219.

Interposition

Adding a sequence of one or more subordinate **Coordinators** between a root **Coordinator** and its participant.

Isolation

A transaction property that allows concurrent execution, but the results will be the same as if execution was serialised. Isolation ensures that concurrently executing transactions cannot observe inconsistencies in shared data.

Life Cycle Object

An object whose interfaces are defined by the Life Cycle Services, specifically remove, copy and move.

Lock Service

Called the Concurrency Control Service, it is an object service used by **Resources** to control access to shared objects by concurrently executing methods.

Log-ready Record (and Contents)

For an intermediate transaction **Coordinator** a log-ready record contains identification of the (superior) transaction **Coordinator** and of **Resource** objects (including subordinate transaction **Coordinators**) registered with the TC which replied **VoteCommit** (that is, it excludes registered objects which replied **VoteReadOnly**); for a **Resource** object a log-ready record includes identification of the transaction **Coordinator** with which it is registered.

Log-commit Record (and Contents)

A log-commit record contains identification of all registered **Resource** objects which replied **VoteCommit**.

Log-heuristic Record

This contains a record of a heuristic decision, either **HeuristicCommit** or **HeuristicRollback**.

Log-damage Record

This contains a record of heuristic damage; that is, where it is known that a heuristic decision conflicted with the decided outcome (**HeuristicMix**) or where there is a risk that a heuristic decision conflicted with the decided outcome (**HeuristicHazard**).

Log Service

An object service used by Resource Managers for recording recovery information and by Transaction Managers recording transaction state durably.

Name Binding

A name-to-object association. A name binding is always defined relative to a naming context.

Naming Context

An object that contains a set of name bindings in which each name is unique.

Naming Graph

A directed graph with nodes and labelled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object.

Nested Transaction

A transaction that either has subtransaction or is a subtransaction on some other transaction.

Participant

See **Commit Participant** on page 219.

Persistent Storage

Generally speaking, a synonym for *stable storage*. In the context of the OMA, the Persistent Object Service provides an object representation of stable storage.

Prepared

The state that a transaction is in when phase one of a two-phase commit has completed.

Presumed Abort

An optimisation of the two-phase commit protocol that results in more efficient performance as the root **Coordinator** does not need to log anything before the commit decision and the participants (that is, **Resource** objects) do not need to log anything before they prepare. So called because, at restart, if no record of the transaction is found, it is safe to assume the transaction aborted.

Propagation

A function of the transaction service that allows the transaction context of a client to be associated with a transactional operation on a server object. The Transaction Service supports both implicit and explicit propagation of transaction context.

Pull Model

An approach to initiating event communication. The pull model allows a consumer of events to request the event data from a supplier. In the pull model, the consumer is taking the initiative.

Push Model

An approach to initiating event communication. The push model allows a supplier of events to initiate the transfer of the event data to consumers. In the push model, the supplier is taking the initiative.

Recoverable Object

An object whose data is affected by committing or rolling back a transaction.

Recoverable Server

An object that registers a **Resource** (not necessarily itself) with a transaction **Coordinator** to participate in transaction completion.

Recovery Service

An object service used by Resource Managers for restoring the state of objects to a prior state of consistency.

Relationship

Relationships allow semantics to be added to references between objects. For example, relationships allow one object to contain another. Life Cycle Services must work in the presence of graphs of related objects.

Resolve

To resolve a name is to determine the object associated with the name in a given context. A name is always resolved relative to a context — there are no absolute names.

Resource

An object in the Transaction Service that is registered for involvement in two-phase commit (2PC). Corresponds to a Resource Manager.

Resource Manager

An X/Open term for a component which manages the integrity of the state of a set of related resources.

Rollback

Rollback (also known as *abort*) has two definitions as follows:

1. An operation in the **Current** and **Terminator** interfaces used to indicate that the current transaction has terminated abnormally and its effects should be discarded.
2. An operation in the **Resource** interface which causes all state changes in the transaction to be undone.

Root Coordinator

The first **Coordinator** in a sequence of **Coordinators** where there is interposition. The **Coordinator** associated with the transaction originator.

Security Service

An object service which provides identifications of users (authentication), controls access to resources (authorisation), and provides auditing of resource access.

Simple Name

A name with a single component.

Stable Storage

Storage not likely to be damaged as the result of node failure.

Sub-coordinator

See **Subordinate Coordinator**.

Subordinate Coordinator

A **Coordinator** subordinate to the root **Coordinator** where interposition exists. A subordinate **Coordinator** appears as a **Resource** object to its superior. Also known as a *sub-coordinator*.

Thread

The entity that is currently in control of the processor.

Thread Service

An object service, to be specified in the future, which enables methods to be executed concurrently by the same process. Where two or more methods can execute concurrently, each method is associated with its own thread of execution.

TP Monitor

A system component that accepts input work requests and associates resources with the programs that act upon these requests to provide a run-time environment for program execution.

Transaction

A collection of operations on the physical and abstract application state.

Transactional Client

An arbitrary program that can invoke operations of many transactional objects in a single transaction. Not necessarily the transaction originator.

Transaction Context

The transaction information associated with a specific thread. See **Propagation** on page 222.

Transactional Operation

An operation on an object that participates in the propagation of the current transaction.

Transaction Originator

An arbitrary program — typically, a transactional client, but not necessarily an object — that begins a transaction.

Transaction Manager

A system component that implements the protocol engine for the two-phase commit protocol. See also **Transaction Service**.

Transactional Object

Strictly speaking, an object that offers at least one transactional operation, and thus requires the ORB and the Transaction Service to propagate transaction context — but usually used to refer to an object none of whose operations are affected by being invoked within the scope of a transaction.

Transactional Server

A collection of one or more objects whose behaviour is affected by the transaction, but which have no recoverable states of their own.

Transaction Service

An object service that implements the protocols required to guarantee the ACID (Atomicity, Consistency, Isolation and Durability) properties of transactions. See also **Transaction Manager**.

Two-phase Commit

A Transaction Manager protocol for ensuring that all changes to recoverable resources occur atomically and furthermore, the failure of any resource to complete will cause all other resource to undo changes. Also called *2PC*.

Typed Event

An event for which an interface is defined in terms of OMG IDL.

Index

2PC.....	219	CosExternalization module.....	86
Abort	219	CosExternalizationContainment module.....	98
Active	219	CosExternalizationReference module.....	99
archive service.....	11	CosGraphs module	135
Atomicity.....	219	CosObjectIdentity module	119
backup/restore service.....	11	CosPersistenceDDO module.....	51
base relationship.....	108	CosPersistenceDS_CLI module.....	54
Begin.....	219	CosPersistencePDS module.....	39
Bind.....	219	CosPersistencePDS_DA module.....	41
callback interfaces.....	7	CosPersistencePID module	29
cardinality		CosPersistencePO module.....	32
maximum.....	119	CosPersistencePOM module.....	35
minimum.....	119	CosReference module.....	143
change management service	12	CosRelationships module.....	120
checked transaction behaviour.....	150	CosStream module.....	89
Commit.....	219	CosTransactions module.....	202
Commit Coordinator	219	CosTSInteroperation module.....	204
Commit Participant.....	219	CosTSPortability module.....	205
Committed	219	Current interface.....	160
Completion	219	Cursor interface	57
Compound Name.....	219	CursorFactory interface.....	57
concurrency control		DA data objects	
basic concepts.....	63	defining and using.....	44
lock modes	66	DAObjectFactory interface	43
locking model.....	66	DAObjectFactoryFinder interface	43
nested transactions.....	69	data interchange service.....	12
possession semantics	67	datastores	54
two-phase transactional locking	68	Datastore_CLI interface	58
concurrency control service	2, 17, 63	DDO protocol	49
Connection interface	56	Decided Commit State	220
ConnectionFactory interface	56	Decided Rollback State.....	220
Consistency.....	220	degree.....	115
containment	142	design decisions	10
context management		design principles.....	5
direct	172	service	5
indirect.....	172	direct access protocol.....	40
Control interface	163	Direct Context Management.....	220
Cooperation	219	Durability	220
Coordinator	220	dynamic data object protocol.....	49
Coordinator interface	165	dynamic state	21
CORBA	220	DynamicAttributeAccess interface.....	46
concepts	5	EdgeIterator interface	141
relationship to	18	Event	220
CosCompoundExternalization module.....	93	Event Channel.....	220
CosConcurrencyControl module.....	70	event service	15
CosContainment module.....	142	exceptions.....	9

- Execution Environment220
- explicit propagation.....172
- externalization
 - client model76
 - interface hierarchy.....82
 - interface summary84
 - NIL data101
 - object hierarchy.....82
 - object model77
 - repeated reference data.....101
 - service structure76
 - specific relationships97
 - standard stream data format100
 - stream model.....76
- externalization service3, 16, 75
- externalize75
- Factory Object220
- failure domain.....191
- Federation220
- FileStreamFactory interface.....87
- Flat Transaction.....220
- flat transactions.....149
- Forgotten State221
- future object services11
- generic DAOObject interface.....42
- global identifier spaces7
- graphs of related objects108
- Heuristic Commit or Abort.....221
- HeuristicCommit exception.....159
- HeuristicHazard exception159
- HeuristicMixed exception159
- HeuristicRollback exception159
- IdentifiableObject interface120
- implementation repository service.....12
- implicit propagation172
- In-doubt.....221
- Indirect Context Management.....221
- intention lock2
- interface
 - inheritance.....9
 - style consistency9
- interface repository service12
- internalization
 - object model79
- internalize.....75
- internationalization service12
- Interposition221
- InvalidTransaction exception.....158
- Isolation221
- Life Cycle Object.....221
- life cycle service15, 18
- local implementations6
- lock.....2
- lock mode
 - intention read66
 - intention write.....66
 - read.....66
 - upgrade.....66
 - write66
- Lock Service.....221
- LockCoordinator interface.....72
- LockSet interface.....72
- LockSetFactory interface.....74
- Log Service.....221
- Log-commit Record (and Contents)221
- Log-damage Record.....221
- Log-heuristic Record.....221
- Log-ready Record (and Contents).....221
- logging service12
- multiple possession semantics67
- Name Binding222
- Naming Context222
- Naming Graph222
- naming service10, 15, 18
- navigation functionality.....118
- Nested Transaction222
- nested transactions.....149
- Node interface.....94, 139
- NodeFactory interface.....140
- non-transactional client.....63
- non-transactional object.....148
- object
 - relationship2
 - role2
 - universal identity10
- Object Model
 - relationship to20
- object transaction service1
- objects conspire6
- ODMG model.....217
- ODMG standard216
- ODMG-93 protocol.....48
- operation mode
 - explicit64
 - implicit.....64
- operations
 - explicit.....9
 - implicit.....9
- ORB interoperability.....18
- OSI TP interoperability213
- other datastores.....61
- other protocols53

Index

Participant	222	consistency constraints	119
PDS		containment.....	97, 142
overview.....	38	CORBA object references.....	105
PDS_ClusteredDA interface.....	46	graph architecture	130
PDS_DA interface.....	43	higher degree.....	115
PDS_DA protocol	40	implementation strategies.....	119
persistent data service		interface summary	112
overview.....	38	key features.....	104
persistent object	28	levels of service.....	108
persistent object service.....	1, 15, 18, 21	operations.....	115, 117
persistent state	21	reference.....	97, 142
Persistent Storage	222	related object graph.....	130, 132
PID interface	29	service structure	108
PIDFactory interface		specific relationships	142
example	30	technical issues	106
PID_CLI interface	57	Relationship interface.....	95, 124
PID_DA interface.....	42	hierarchy	111
PO interface	32	relationship service.....	2, 15, 18, 103
POFactory interface.....	34	relationship type.....	114
POS	1, 21	RelationshipFactory interface	123
basic capabilities.....	23	RelationshipIterator interface	129
goals	23	remote implementations	6
object-oriented storage.....	23	replication service.....	14
open architecture.....	24	Resolve.....	223
properties	23	Resource	223
service structure	27	Resource interface	168
views of service.....	25	Resource Manager.....	223
Prepared	222	return codes	9
Presumed Abort.....	222	role	104
Propagation.....	222	Role interface	95, 125, 140
PropagationCriteriaFactory interface.....	96	hierarchy	111
Pull Model.....	222	RoleFactory interface	128
Push Model.....	222	Rollback	223
quality of service.....	6	Root Coordinator.....	223
query service.....	13	SD interface.....	34
recoverable object	148	security service.....	14
Recoverable Object.....	222	Security Service.....	223
recoverable server	149	service	
Recoverable Server.....	222	basic flexible	5
recovery service	14	concurrency control	2
Recovery Service.....	222	dependencies.....	15
RecoveryCoordinator interface	168	externalization.....	3
reference	142	generic.....	5
related object	114	object transaction.....	1
relationship.....	104	persistent object	1
Relationship.....	223	relationship	2
relationship		transaction.....	1
attributes	115	services	
base model.....	114	finding.....	8
binary.....	115	using.....	8
compound operations	133	Simple Name	223

SNA LU 6.2 interoperability.....	214	standard exceptions.....	158
specific relationship.....	108	TP standards.....	207
Stable Storage.....	223	typical usage.....	156
standards		unchecked services.....	178
conformance.....	20	user view.....	172
standards conformance.....	61	Transaction Context.....	224
startup service.....	14	Transaction Manager.....	224
Stream interface.....	87	transaction originator.....	148, 150
Stream service.....	76	Transaction Originator.....	224
Streamable interface.....	91	transaction service.....	1, 16, 18, 145
StreamableFactory interface.....	92	Transaction Service.....	224
StreamFactory interface.....	86	transactional client.....	63
StreamIO interface.....	90	Transactional Client.....	224
Sub-coordinator.....	223	transactional object.....	148
Subordinate Coordinator.....	223	Transactional Object.....	224
SubtransactionAwareResource interface.....	170	Transactional Operation.....	224
Terminator interface.....	164	transactional server.....	149
Thread.....	223	Transactional Server.....	224
Thread Service.....	223	TransactionalLockSet interface.....	73
TP Monitor.....	223	TransactionalObject interface.....	171
Transaction.....	224	TransactionFactory interface.....	163
transaction		TransactionRequired exception.....	158
application portability.....	177	TransactionRolledBack exception.....	158
application programming models.....	172	traversal criteria	
applications.....	146	example.....	134
checked behaviour.....	175	Traversal interface.....	138
checked services.....	178	TraversalCriteria interface.....	138
context.....	156	TraversalFactory interface.....	137
context management.....	157	Two-phase Commit.....	224
data types.....	157	TX interface.....	207
definitions.....	147	TX mappings.....	207
design principles.....	151	Typed Event.....	224
distributed transactions.....	177	unchecked transaction behaviour.....	150
examples.....	178	UserEnvironment interface.....	56
exceptions.....	158	WrongTransaction exception.....	159
failure models.....	183	X/Open Resource Manager.....	209
function principles.....	151	XA mappings.....	209
functionality.....	149	XA-compliant Resource Manager.....	210
heuristic completion.....	176	XID.....	209
heuristic exceptions.....	158		
implementor view.....	185		
interfaces.....	160, 174		
interoperation.....	213		
model interoperability.....	180, 200		
ORB/TS.....	194		
other exceptions.....	159		
overview.....	145		
performance principles.....	151		
recoverable server.....	176		
service architecture.....	155		
service protocols.....	185		