

**INTEGRATING EXISTING OBJECT-ORIENTED DATABASES
WITH DISTRIBUTED OBJECT MANAGEMENT PLATFORMS**

- Developed and evaluated on the example of
ODBMS ObjectStore and CORBA -

Dissertation zur
Erlangung des Doktorgrades (Dr. rer. nat.) der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Sergey Shumilov

Bonn
2003

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelm-Universität Bonn

Abstract

A common characteristic of today's information systems is the distribution of data among a number of autonomous systems and heterogeneous repositories. The key to their unification is the ability to access these multiple databases transparently, irrespective of their kind and location. However, direct database-level interoperation is not always possible because of the extreme heterogeneity of already existing applications, software environments and hardware platforms. A necessary infrastructure to deal with the heterogeneity and remote access in the object-oriented context can be provided by modern distributed object management platforms, such as a Common Object Request Broker Architecture (CORBA).

The thesis puts forward the concept of a framework for providing object persistence in object-oriented programming languages by integrating the powerful database facilities with the heterogeneous CORBA environment – the *eXtensible Database Adapter (XDA)*. Providing a highly customisable environment with pre-defined *adaptable, ready to use components* the framework simplifies the development of the middle-tier level mediating between a database and CORBA. The conceptual basis of the framework is a *distributed persistent object model* that on the one hand provides the property of *transparent access to persistent objects* for usual CORBA objects and, on the other hand allows a *soft, non-intrusive integration* for existing databases. Thus, the solution possesses various valuable advantages such as the facility of creating high performance CORBA/ODBMS systems coupled with maximal flexibility, as well as the ability to build new, smoothly integrated, legacy database applications within the CORBA environment.

The vitality of the proposed concept is demonstrated on two implementations of the XDA, which has allowed us to identify the crucial problems appearing in the integration process and to propose corresponding solutions. Implementing new approaches for *efficient request dispatching* and *transaction management* the XDA permits the development of efficient and scalable distributed systems dealing with large numbers of persistent objects. Additionally, two alternative approaches intended to simplify the development of CORBA-adapters are studied: *dynamic mediators* and *automated code generation*.

At the time of writing, a running prototype of the XDA developed as part of this thesis supports the ObjectStore database management system. This prototype has been successfully applied in several geological research projects concerning the development of *large 3D kinematic models* of selected regions within the Lower Rhine Basin. We expect that the enhanced facilities in handling large sets of complex persistent data provided by techniques presented will enable the creation more comprehensive and powerful distributed database management systems.

Table of Contents

ABSTRACT	iii
TABLE OF CONTENTS.....	v
LIST OF FIGURES	ix
LIST OF TABLES	xiii
ACKNOWLEDGMENTS	xv
ABBREVIATIONS	xvii
CHAPTER 1	
INTRODUCTION.....	1
CHAPTER 2	
BASICS OF OBJECT-ORIENTED DATABASES AND CORBA	9
2.1 Object-Oriented Database Management Systems	9
2.1.1 ODMG Standard	9
2.1.2 ObjectStore as an Example of ODBMS	11
2.2 The Common Object Request Broker Architecture.....	18
2.2.1 Why CORBA?	18
2.2.2 The Object Management Architecture	19
2.2.3 Object Request Broker.....	19
2.2.4 CORBA Application Development Steps.....	21
2.2.5 The Interface Definition Language (IDL).....	21
2.2.6 Operation Invocation and Dispatching.....	24
2.2.7 Portable Object Adapter (POA)	28
2.2.8 CORBA Implementations: Orbix and ORBacus.....	31
2.3 Motivations to CORBA/ODBMS Integration.....	34
2.3.1 Motivations of CORBA.....	34
2.3.2 Motivations of ODBMSs	36
2.3.3 The Benefits of CORBA/ODBMS Integration	39
CHAPTER 3	
THE EVOLUTION OF CORBA/ODBMS INTEGRATION TECHNIQUES	41
3.1 Basic Integration Strategies	41
3.1.1 Server-side Query Approach.....	41
3.1.2 Object Collections.....	42
3.1.3 Pure CORBA	42
3.2 Wrapping persistent Objects	43
3.3 Object Database Adapters (ODA).....	44
3.3.1 Early ODAs	44
3.3.2 Orbix Database Adapter Framework.....	45
3.3.3 Front-End Generators	51
3.3.4 Approaches for Transaction Management.....	51
3.4 Persistent CORBA Services.....	52
3.4.1 Differences between the POS and the PSS.....	53

Table of Contents

3.4.2	Overview of the PSS.....	53	
3.4.3	Drawbacks of the PSS	56	
3.5	Other CORBA Services.....	57	
3.5.1	The Object Query Service (OQS).....	57	
3.5.2	The Object Transaction Service (OTS).....	57	
3.5.3	Solutions based on OTS	57	
3.6	Shortcomings of the existing CORBA/ODBMS Integration Tools	59	
CHAPTER 4			
DESIGN OF THE eXTENSIBLE DATABASE ADAPTER.....			61
4.1	Overview of the Approach	61	
4.1.1	The Distributed Persistent Object Model	61	
4.1.2	The eXtensible Database Adapter (XDA).....	62	
4.2	General Design Issues	63	
4.2.1	Using Mediators as Intermediate Communication Components.....	64	
4.2.2	Three Tier Architecture.....	65	
4.2.3	Transparent Persistence	66	
4.3	The Distributed Persistent Object Model	67	
4.3.1	Abstract Model	68	
4.3.2	Mapping of persistent C++ to IDL.....	72	
4.3.3	Execution Model.....	88	
4.4	eXtensible Database Adapter Framework.....	98	
4.4.1	Architecture of the XDA Application Server	99	
4.4.2	The XDA Framework Concepts.....	99	
4.4.3	Developing integrated CORBA/ODBMS Applications	106	
4.4.4	Comparing XDA-based and native ObjectStore Applications.....	108	
CHAPTER 5			
IMPLEMENTATION OF THE XDA FRAMEWORK			111
5.1	The XDA Architecture	111	
5.2	Implementing Mediator Components.....	112	
5.3	Request Dispatching and Mediator Management.....	113	
5.3.1	Interoperable Persistent Object References.....	113	
5.3.2	Activation.....	114	
5.3.3	Deactivation	117	
5.4	Transaction Management	118	
5.4.1	Demarcation and Management of Transactions in XDA	118	
5.4.2	Manual Mode	119	
5.4.3	Automatic Mode.....	120	
5.4.4	Performance Improvements	122	
5.5	XDA's CORBA Interface	123	
5.5.1	Adapter Interface.....	124	
5.5.2	Database Interface.....	124	
5.5.3	Object Interface.....	125	
5.5.4	Type Interface	125	
5.5.5	Transaction Interface.....	126	
5.5.6	Collections Interface.....	126	
5.6	The Spatial Object Database Adapter (SODA)	129	
5.6.1	Implementing custom Managers	129	
5.6.2	Implementing custom Mediators	130	

5.6.3	Implementing abstract Mediators	130
5.6.4	Issues with Multiple Inheritance and Downcasting.....	131
5.6.5	Evaluation of the BOA-based XDA Implementation.....	132
CHAPTER 6		
IMPROVING THE XDA FRAMEWORK		135
6.1	Using the Portable Object Adapter.....	135
6.1.1	Advantages of the Portable Object Adapter.....	135
6.1.2	Architecture of the POA-based XDA	137
6.2	New Techniques for the Implementation of Mediators	138
6.2.1	Implementing complex Static Mediators.....	138
6.2.2	Dynamic Mediators and Metaobject Protocol Interface	145
6.2.3	Generation of Mediators	149
6.3	New Techniques for persistence-aware Request Dispatching	151
6.3.1	Performance and Scalability Problems of common Techniques	151
6.3.2	Using multiple POAs for more efficient Request Dispatching.....	153
6.3.3	Using POA Servant Managers for Request Dispatching.....	154
6.3.4	Persistence-aware Method for Request Dispatching.....	157
6.3.5	Efficient Eviction Technique based on Reference Substitution	159
6.3.6	Efficient Mediator Replacement Policies	161
6.3.7	Evaluation of the new Dispatching Technique	176
6.4	New Techniques for Transaction Management.....	180
6.4.1	Problems with Automatic and Manual Transactions	181
6.4.2	Multiclient Concurrency Control	182
6.4.3	Managing available Resources.....	187
6.4.4	Chained Transactions	190
6.4.5	Evaluating the proposed Techniques	192
6.5	Designing robust CORBA/DBMS Applications	195
6.5.1	Design of IDL Interfaces	195
6.5.2	Performance Issues.....	196
6.5.3	Using Patterns to Build an Efficient CORBA/ODBMS System	200
CHAPTER 7		
CASE STUDY: A DISTRIBUTED 3D/4D GIS.....		203
7.1	Motivation	203
7.1.1	Examples of geological 3D/4D Models.....	204
7.1.2	Requirements for the Storage and Retrieval of spatio-temporal Objects.....	205
7.2	CORBA-based integrated Architecture.....	206
7.2.1	Spatio-temporal object-oriented Database.....	206
7.2.2	GeoToolKit/CORBA Adapter	207
7.2.3	Portable Client Interface.....	207
7.2.4	Transmission of large and complex Objects.....	211
7.3	Performance Evaluation.....	213
7.3.1	Benchmark Description	213
7.3.2	Test Environment.....	215
7.3.3	Basic Experimental Results	216

Table of Contents

CHAPTER 8	
CONCLUSIONS AND FURTHER RESEARCH	223
8.1 Thesis Summary	223
8.2 Applicability of the XDA Approach	225
8.2.1 Opening legacy Systems	225
8.2.2 CASE and CAD Systems.....	225
8.2.3 Geographic Information Systems	226
8.2.4 Information Systems for Medicine and Biology.....	226
8.2.5 Telecommunications Management Networks	227
8.3 Perspectives and Future Work.....	227
BIBLIOGRAPHY.....	231

List of Figures

Number	Page
Figure 1. The hierarchical relationship among allocation units in ObjectStore	14
Figure 2. Schema of the typical networked ObjectStore application	16
Figure 3. The Object Management Architecture	18
Figure 4. The main components of the CORBA	19
Figure 5. CORBA application development process	21
Figure 6. Basic request dispatching in CORBA	24
Figure 7. A request invocation	25
Figure 8. Operation dispatch in static invocation.....	26
Figure 9. Inheritance-based approach.....	26
Figure 10. Delegation-based approach	27
Figure 11. Mixing of static and dynamic interfaces	27
Figure 12. Request dispatching in POA	30
Figure 13. The schema of object loading and activation at runtime	32
Figure 14. Three-tier architecture of the CORBA/ODBMS system.....	34
Figure 15. Comparison of client/server architectures of RDBMS, ODBMSs and CORBA..	37
Figure 16. The functional schema of the wrapping technique.....	43
Figure 17. The functional schema of the integration using the Object Database Adapter	44
Figure 18. The functional schema of the integration using the ODAF-made adapter.....	46
Figure 19. TIE object creation process.....	48
Figure 20. Per-operation transaction style.....	48
Figure 21. Phased transaction style.....	49
Figure 22. ODAF-supported approaches for realising persistent objects	50
Figure 23. PSS Basic Concepts.....	54
Figure 24. The CORBA/ODBMS integration architecture	58
Figure 25. Mediators as intermediate communication components	64
Figure 26. Three Tier Architecture of CORBA/ODBMS system.....	65
Figure 27. Three Scenarios for Object By Value	87
Figure 28. Type and Classes.....	88
Figure 29. Binding of mediators to database objects	89
Figure 30. Interoperable Persistent Object Reference (IPOR)	92
Figure 31. The functional architecture of the application server	99
Figure 32. XDA's component-based architecture	101
Figure 33. Mediator management	103
Figure 34. An example of the nested transactions.....	106
Figure 35. CORBA→ODBMS application development steps	107
Figure 36. ODBMS→CORBA application development steps	108

List of Figures

Figure 37. The XDA class diagram.....	112
Figure 38. Implementation of the user-defined mediator	113
Figure 39. Implementation of IPOR.....	114
Figure 40. Request dispatch in the BOA-based XDA	114
Figure 41. Hierarchical organisation of dictionaries for back references	115
Figure 42. Request dispatching for persistent objects.....	116
Figure 43. Reference resolving for input/return parameters in a mediator.....	116
Figure 44. Example of the CORBA client code that performs explicit deactivation of mediators	117
Figure 45. Example of the CORBA server code that performs the demarcation of a declarative „update“ transaction	119
Figure 46. Demarcation of transaction boundaries during the request processing.....	121
Figure 47. Persistence checking and transaction type setting mechanisms.....	123
Figure 48. Example of manipulation with databases.....	125
Figure 49. Example of the client code that uses different transaction control methods.....	126
Figure 50. Example of the IDL mapping for collections using the value-based approach ...	127
Figure 51. Example of the IDL mapping for collections using the IDL interface approach	127
Figure 52. Using <code>XDA_Cursor</code> for iterating through an unordered collection	128
Figure 53. Architecture of the <i>Spatial Object Database Adapter</i> (SODA)	129
Figure 54. Class diagram of the <i>Spatial Object Database Adapter</i>	130
Figure 55. Extending the XDA with a new mediator class.....	130
Figure 56. Extending the XDA with a new class for database entities	131
Figure 57. An example of multiple inheritance between database classes	131
Figure 58. Class diagram of the POA-based XDA.....	137
Figure 59. An inheritance hierarchy for classes implementing collection interfaces.....	138
Figure 60. Separating inheritance hierarchies for interfaces and mediator classes.....	139
Figure 61. Implementation of the static mediators in the POA-based XDA.....	140
Figure 62. Implementing mediators for database classes with multiple inheritance	141
Figure 63. Implementing mediators for two classes from ObjectStore’s collection library...	142
Figure 64. An example of possible correspondence between IDL interfaces, mediators and database classes.....	142
Figure 65. An example of implementing mediators for two IDL interfaces	143
Figure 66. A general schema for implementing mediators supporting multiple interfaces...	143
Figure 67. A general schema for implementing mediators for composite types.....	144
Figure 68. An example of implementing mediators for two composite types	145
Figure 69. Architecture of the dynamic database interaction	146
Figure 70. General schema for the implementation of static and dynamic mediators	147
Figure 71. Building mediators with the help of the XDA Code Generator	150
Figure 72. Using several POA adapters for more efficient request dispatching	153
Figure 73. Format of the Interoperable Persistent Object Reference (IPOR).....	154
Figure 74. Class diagram for the XDA ObjectManager	155
Figure 75. Control flow diagram of request dispatching for user-defined database objects in the POA-based XDA.....	156

Figure 76. Comparison of the standard POA and XDA schemas of request dispatching for user-defined database objects	157
Figure 77. Performance of hash tables on keys of different size: I – search for a key, II – search and replacement of a key (search /remove / insert)	159
Figure 78. Request dispatching in the POA-based XDA using the reference substitution technique	160
Figure 79. Times for the eviction of mediators depending on the number of associated database objects.....	160
Figure 80. Hit ratio of the four groups of traces under LRU, Size, LRV, GD-Size(1), and GD-Size(packets) [CI97]	167
Figure 81. An evictor queue after instantiating five mediators and after eviction of the mediator „1“	169
Figure 82. Top curve: the probability of more accesses after i previous ones ($P(i)$). Middle curve: $P(i)$ computed without considering new accesses issued after 15 days. Bottom curve: percentage of documents with at least i accesses [RV00]	171
Figure 83. The curves represent probabilities that the object will be reaccessed after f_o previous ones – $P(f_o)$ and after time t_o – $P(t_o)$. The bottom curve is the product of the both functions P for $t_o=f_o$	175
Figure 84. Performance of particular operations for BOA- (I) and POA-based XDAs (II) with unlimited cache	178
Figure 85. Performance of request dispatching in BOA- and POA-based XDAs for different cache sizes: I – average call latency for operation ping (second), II – average time for a total sequence of operations (create, retrieve, ping, ping, delete).....	179
Figure 86. Memory consumption in BOA- and POA-based XDAs for different cache sizes. Figure I represents a zoomed part of figure II in the region 0..10000	180
Figure 87. Using notifications for controlling delayed transactions	185
Figure 88. Sequence diagram for an example of using notifications presented in Figure 87	185
Figure 89. Organisation of memory in ObjectStore applications	188
Figure 90. Performance of the test application using different transaction styles: I – <i>manual</i> , II – <i>perOp</i> , III – <i>delayed</i> (1trx), IV – <i>delayed</i> (1trx, checks)	193
Figure 91. Performance of particular operations using different transaction styles: I – <i>set_id</i> (second), II – <i>multiply_write_good</i>	194
Figure 92. Total performance of all operations of the test application using different transaction styles	194
Figure 93. Performance comparison of „bad“ and „bulk“ implementations for <i>insert()</i> operation inserting the data of different size – 100*4 and 200*4 bytes	198
Figure 94. Two geological models in the 3D viewer of GeoToolKit	204
Figure 95. Structure of the Bergheim geometry model	205
Figure 96. CORBA-based integrated architecture of the prototype	206
Figure 97. Architecture of the client and extension of a VRML-browser for visualisation of time-dependent spatial data	207
Figure 98: A screenshot of the client’s graphical user interface	209
Figure 99. 4D geometry objects at different time steps - above: 18.5 million years ago, below: today - in a VRML browser (in our prototype - Cortona™ [Par02]).....	210
Figure 100. Architecture for progressive transmission	212
Figure 101. IDL interface of the test application.....	213

List of Figures

Figure 102. Navigation schemas of „good“ and „bad“ multiplication operations	214
Figure 103. Test environment.....	215
Figure 104. Performance of simple operations taking one parameter (operations <code>set_id</code> for objects of size 100*4 bytes – I, and 200*4 bytes – II)	216
Figure 105. Performance of operations taking „bulk“ parameters (operations <code>set_data</code> for objects of size 100*4 bytes – I, and 200*4 bytes – II)	217
Figure 106. Performance of data-intensive operations (I – operations <code>multiply_read/write_good</code> , II – operations <code>multiply_read/write_bad</code>)....	218
Figure 107. A stratigraphic surface with reduction factors: 0%, 60% and 80%	219
Figure 108. Part of the stratigraphic surface with reduction factors: 0% and 60%.....	220
Figure 109. Speed of data transfer in relation to the object size	221
Figure 110. Past and predicted evolution of CORBA/ODBMS tools.....	228

List of Tables

Number	Page
Table 1. Comparison of page and object servers.....	15
Table 2. Examples of IDL to C++ language mapping.....	24
Table 3. C++ to IDL mapping for names.....	72
Table 4. C++ to IDL mapping of namespaces.....	73
Table 5. C++ to IDL mapping for basic types.....	73
Table 6. C++ to IDL mapping for constants.....	74
Table 7. C++ to IDL mapping for constructed and user-defined types.....	74
Table 8. C++ to IDL mapping for classes.....	75
Table 9. C++ to IDL mapping for overloaded operations.....	76
Table 10. C++ to IDL mapping for operators.....	77
Table 11. C++ to IDL mapping for static class members.....	79
Table 12. C++ to IDL mapping for pointers passed by reference.....	80
Table 13. C++ to IDL mapping for pointers passes by value (struct-based representation)...	82
Table 14. Comparison between features of database and CORBA clients.....	110
Table 15. Tradeoffs between different representations of collections.....	129
Table 16. Values of $K(k_1, k_{max}, f_{max})$ for some points.....	172
Table 17. Comparison of the standard Evictor pattern with the proposed method for reference substitution.....	177
Table 18. Schedules for usual and MVCC transactions in the simple waiting scenario.....	183
Table 19. Schedules for usual and MVCC transactions in the simple deadlock scenario.....	183

Acknowledgments

This thesis would have been impossible without the support and encouragement of many people. I would like to use this opportunity to express my deepest gratefulness to all of them.

First of all, I would like to thank my scientific advisor Prof. Armin B. Cremers for his support, inspiring comments and patience during the research and completion of this thesis.

I would also like to thank my former colleagues Prof. Martin Breunig, Dr. Oleg Balovnev and Andreas Bergmann. Their deep, stimulating insights and valuable remarks provided crucial help with the thesis, not to mention the excellent research environment, which would have been impossible to obtain without their assistance.

I sincerely appreciate the aid of Dr. Thomas Bode, who provided helpful comments and suggestions, and with whom I was fortunate to share the same room at the University. Many thanks go to my students Bjorn Koos, Marcus Baeurle and Muhammed Tasci for our valuable discussions, which contributed many interesting ideas to my work.

The concepts realised in the present work have been developed in close cooperation with the Geological Institute at the University of Bonn. Our thanks go to Prof. Agemar Siehl and Andreas Thomsen who provided the necessary application background and numerous useful remarks. Our further thanks go to RWE Rheinbraun AG, Cologne and its chief mining surveyor, Sven Asmus, for providing a series of large real world geometric objects for testing purposes.

And last, but surely not least, I would like to thank the Marina for her affection, care, and support given to me when it was most needed. Without this support my work would be extremely hard. Finally Tom Arbuckle for help with English and corrections.

Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DDL	Data Definition Language
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
DTP	Distributed Transaction Processing
CG	Code Generator
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
MOP	Metaobject Protocol
MVCC	Multiversion Concurrency Control
NPT	Non-Persistent TIE
OBV	Objects By Value
ODA	Object Database Adapter
ODAF	Orbix Database Adapter Framework
ODBMS	Object Database Management System
ODL	Object Definition Language
ODM	Object-Oriented Data Model
ODMG	Object Database Management Group
OMA	Object Management Architecture
OMG	Object Management Group
OOSA	Orbix+ObjectStore Adapter
OQL	Object Query Language
ORB	Object Request Broker
OSSG	ObjectStore Schema Generator
OTS	Object Transaction Service
PDT	Programmer-Defined TIE
PPL	Persistent Programming Language
POA	Portable Object Adapter
POS	Persistence Object Service
PSS	Persistent State Service
PSDL	Persistent State Definition Language
RDBMS	Relational Database Management System
SIGDB	Special Interest Group on Object Databases
SQL	Standard Query Language
VMMA	Virtual Memory Mapping Architecture
XA	The X/Open DTP resource manager interface
XDA	The eXtensible Database Adapter
XML	The eXtensible Markup Language
XODL	The eXtensible Object Definition Language

Chapter 1

Introduction

Towards open heterogeneous distributed Systems

All computer systems have their limitations. These limitations can be seen in the amount of memory a system can address, the number of hard disk drives which can be connected to it or the number of processors it can run in parallel. In practice this means that, as the quantity of information in a database becomes larger, a single system can no longer cope with all the information that needs to be stored, sorted and queried.

Although it is (currently) still possible to build bigger and faster computer systems, it is often not a cost-effective solution to upgrade the hardware every few months. Instead, it is more affordable to have several database servers that appear to the users as a single system, and which split the tasks between themselves. By doing this we can use commodity machines at affordable prices. This has the additional advantage that systems are not simply discarded as soon as a newer version arrives, but can be added to and then replaced as they become obsolete. These are called *distributed information systems* and have the common characteristics that they are stored on two or more computers, called nodes, and that these nodes are connected over a network.

For many applications, distributed information systems are generally understood to provide greater performance and survivability than their centralised counterparts (e.g. [OV91]). For example, the fast growing number of spatio-temporal applications in fields such as environmental monitoring, geology and mobile communication present new challenges to the development of distributed information systems. Due to the greater complexity and large amount of interactive work, the need for distributed database management systems arises frequently in the domain of managing and analysing subsurface 3D geometry data in geology. On the one hand, the introduction of two additional dimensions causes an enormous increase in the volume of raw data, resulting in the need for their compact storage and transmission. On the other hand, the analysis of geo-scientific data by means of different problem-specific tools requires the use of the interactive query processing facilities of the database ranging from complex set-based operations such as spatio-temporal joins to simple selections for on-line visualisation [BBCS00c]. Most modern 3D modelling tools do not have these features and it is not possible to employ them directly for the construction of complex large spatial models [STCK02].

Application Requirements

In the above mentioned application context, several important requirements emerge.

Persistence: Design objects must persist between work sessions.

Distribution: Access must be distributed in order to support geographically-spread work teams.

Concurrency and consistency: Concurrent access allows engineers to work on the same data at the same time. They may explore conflicting alternatives despite this concurrent access. However, data must be kept consistent.

Scalability: Interactive applications such as Geo-Information Systems (GIS) need high performance, even when the number of users or data objects is high and when users are geographically distant from each other.

Security: Data access should be controlled, and its transport should be safe.

Fault tolerance: The platform must prevent the loss of data, e.g. if the system crashes in the middle of a long working session.

In short, our application area requires efficient sharing mechanisms, providing direct access to remote data. As we will see, existing distributed object environments are not adequate.

Modern distributed Information Systems

During recent decades, the realm of database management systems (DBMS) has been unquestionably dominated by relational systems (RDBMS). However, taking into account the complex structure of spatial data, the relational data model has proved to be unsuitable for the presented requirements [BBC97]. After investigating various alternatives, it became evident that a number of the requirements could be met by full object-oriented database management systems (ODBMS) – as defined by the „Object Database Manifesto“ [SRL+90] – and perhaps not even by these. Indeed, many „simpler“ alternatives, such as language extensions or lightweight storage managers, lacked essential functionality – support for heterogeneity, for example.

The appropriate infrastructure to build distributed heterogeneous systems is provided by modern middleware platforms, and one of the most prominent examples is the Common Object Request Broker Architecture (CORBA) of the OMG [OMGorb01]. CORBA defines an object-oriented environment for the interconnection and interoperation of diverse software components. However, protecting data integrity and constructing reliable (distributed) applications is not possible without such concepts as transactions and other facilities, which are typical for database management systems.

At the same time, since both the ODBMS and CORBA technologies are relatively young, they still have some deficiencies, preventing many software designers from using CORBA in mission-critical applications as well as leading to the inability of ODBMSs to finally conquer relational predecessors. However, their integration into one system can eliminate, or at least substantially alleviate, many of those limitations. Through the integration with ODBMSs, modern CORBA tools can be provided with database-specific features like persistence, concurrency control, data recovery, ACID transactions, etc. In turn, the integration with the CORBA environment allows ODBMSs to essentially enhance the potential set of database clients providing remote data access from many heterogeneous platforms, better authorisation mechanisms, implementation of database views, etc. Thus, the profits made by ODBMSs from the integration are even more substantial and diversified than those for CORBA.

Efforts for Integration of ODBMS and CORBA

From the beginning of their existence, both technologies' tendencies towards integration were visible. In particular, in 1993, within the framework of the OMG, a *Special Interest Group on Object Databases* (SIGDB) was created. Later it was given the name *Object Database Management Group* (ODMG) – a consortium associating all major vendors of ODBMS. This group has tried to define a standard, allowing database source code portability. It outlined the principles of the standard-compliant interoperation through the CORBA [CB99].

Similar efforts were also made by middleware systems. The OMG made vigorous attempts to standardise domain-independent means of providing database capabilities for distributed CORBA objects [OMGpos94, OMGpos00]. Regardless of those considerable undertakings, no viable standard of CORBA and ODBMSs interoperation has been specified. Instead, the ODMG and OMG only put forward, without providing any further details, some recommendations concerning a special-purpose *Object Database Adapter* (ODA) residing between CORBA and ODBMS tools and coordinating their operation. The lack of a well-specified standard has resulted in disorganised efforts of CORBA and ODBMS vendors to provide proprietary tools supporting the CORBA/ODBMS integration. Some vendors have

released proprietary ODAs strictly oriented towards the development of new CORBA-compatible database applications. As a rule, those products offer limited flexibility and have rather poor performance. To overcome those limitations, some users are forced to implement in-house software layers wrapping specific databases to make them accessible via CORBA. Although such a strategy offers much more flexibility and is applicable to the integration of pre-existing database applications within a CORBA environment, its realisation requires much in the way of programming and is not easy.

Despite the significant benefits the CORBA/ODBMS integration brings to both parties, little has been published about this subject (among few examples is [Baker96]). Certain research has been conducted in some academic centres (e.g. [RM97, DD98, Ami98, SLY+99]), and by several CORBA and ODBMS vendors ([IToosa97, ITodaf97, ITova97, ITo296, Prism00]).

Goals of the Thesis

The main goal of the thesis is development of a model for providing object persistence in object-oriented programming languages by integrating the powerful database facilities with the heterogeneous CORBA environment. The target *distributed persistent object model* must, on the one hand, provide the property of *transparent access to persistent objects* for usual CORBA objects and, on the other hand, allow a *soft, non-intrusive integration* for existing databases. Thus, the solution should possess various valuable advantages such as the facility of creating high performance CORBA/ODBMS systems coupled with maximal flexibility, as well as the ability to build new, smoothly integrated, legacy database applications within the CORBA environment.

The vitality of the proposed concept should be demonstrated by implementation of the *eXtensible Database Adapter* (XDA). The XDA will implement the proposed model and provide a highly *customisable framework* with pre-defined *adaptable, ready to use components* simplifying the development of the middle-tier level mediating between a database and CORBA. Further, to support the development of efficient and scalable distributed systems dealing with large numbers of persistent objects, corresponding solutions for *efficient request dispatching* and *transaction management* for persistent objects should be proposed. Additionally, alternative approaches that could simplify the interaction with *dynamically changing persistent data stores* should be studied. Finally, the proposed XDA framework will be tested under construction of the system linking some *real pre-existing geoscientific data stores* within a heterogeneous distributed CORBA environment.

Design Issues

In order to realise the thesis goal, the following design issues have to be solved.

First, eliminating the gap between distributed (OMG) and database data models (ODMG), a model that provides a *single paradigm* for designing distributed persistent objects must be developed.

Database programming language syntax to IDL back-mapping. The structure of objects stored in a database should be translated to the CORBA language-neutral Interface Definition Language (IDL). That is to say that the *database programming language syntax to IDL back-mapping* has to be performed. As IDL serves only for the specification of an object's interface. There are some essential inherent differences between IDL and database programming languages such as C⁺⁺. In particular, IDL lacks any means of describing the implementation of individual objects. Thus, the mapping from C⁺⁺ to IDL in some cases can result in a significant loss of information.

Call by value vs. call by reference. Additional efforts should be made to keep the logic and layouts of the original database structures during the transformation to IDL. In particular, the choice of the call semantic (*call by value* vs. *call by reference*) can have a

great influence on the performance of the distributed program. Additionally, since in CORBA the objects are always passed by reference, a mechanism supporting passing *objects by value* should be designed.

Transparent persistence. There should be no difference for the CORBA client in manipulating either persistent or transient objects. For „pure“ CORBA clients the persistence of processed objects has to be hidden, i.e. *transparent*.

Persistence of object references. It should be also noted that persistent CORBA references must be valid for substantially longer times than usual CORBA references. The references to persistent objects can be stored by the clients locally or exchanged.

The fact that we have to deal with *large data stores* that are permanently in use by multiple database applications imposes additional requirements on the interoperation architecture.

Soft, non-intrusive integration for existing databases. Since database scheme evolution may be extremely time-consuming, CORBA conformity must be achievable *without any modifications* to existing databases and applications. It is also important that these new databases, which are remotely accessible through CORBA, at the same time *remain compatible* with native database applications.

Server memory management. One of inherent problems of CORBA (as well as other distributed object-oriented platforms) is *garbage collection* of unused server objects. Disposal of the objects is necessary in order to prevent memory leaks and improve scalability of the integrated system. Since CORBA clients have no control of server objects, the problem of server memory management must be resolved by some additional mechanisms.

The infrastructure must *simplify* development of CORBA/ODBMS applications providing the necessary *guidelines* and *pre-defined adaptable components* for building *scalable, well-performing* integrated systems.

IDL views. Lack of support for views in ODBMSs can be alleviated through „IDL views“. Indeed, since CORBA clients communicate with server objects exclusively via defined IDL interfaces, the „image“ seen by the clients can be effectively handled through the IDL interfaces. The adapter, however, must provide necessary facilities for implementation of such objects and be able to make their *unique identification*.

The structure of „view“ objects should be easily *manageable* and *adaptable* to particular application requirements and schema changes. CORBA server objects interacting with underlying database should be *resistant* to schema changes. Special attention should be paid here to providing convenient means for *customisation* of the code. Thus, *dynamically adaptable* interaction components are required.

Conversion of object references. The integration infrastructure should provide a service for efficient resolution of *out-going database references* in CORBA references. Similar conversion is necessary for all *incoming CORBA references* that must be converted to the corresponding database references before they can be used in database operations.

Scalable request dispatching. Memory consumption and, thus, the scalability of the CORBA/ODBMS environment depend on the number of CORBA server objects, which are used for request dispatching. The *efficient caching* of server objects in a transactional CORBA/ODBMS environment has, however, not been studied in the literature so far. The problem addressed in this thesis is how to design request dispatching and object caching in such an integrated system, while considering *performance* and *scalability* issues.

Finally, a CORBA/ODBMS integrated system should be tuned to provide the best potential *performance*, since in large-scale distributed systems, network communication and database access can degrade performance significantly.

Reasonable granularity of remote invocations. While CORBA clients access objects via long remote method invocations, access to these objects is much slower than to the local cache, which would be used in this case by database management systems. Thus, for distributed applications accessing a large number of *fine-grained* CORBA objects, this overhead would be unacceptable. For these reasons, some convenient means of adjusting the *composition* of the resulting CORBA/ODBMS system, e.g. by allowing the selection of a suitable subset of database objects to be accessed by CORBA clients, should be provided.

Efficient persistence-aware request dispatching. The performance of request dispatching has traditionally been achieved by using hash tables, e.g. in the development of operating systems. The default request dispatching provided by CORBA, however, does not take into account the complex nature of persistent objects. Therefore, more efficient *persistence-aware methods* for request dispatching are necessary.

Efficient transaction control. Transaction management is one of the most crucial points here because any access to persistent data should be made within a corresponding transaction. On the one hand, transparency of objects assumes transparency of other database facilities, for example, in providing efficient mechanisms for *automatic transaction control*. On the other hand, for advanced clients, the adapter must also provide the possibility of *customising the behaviour* of these mechanisms and access to the *native database functionality*, e.g. to control the management of databases, transactions and the clustering of persistent objects.

The Scope of the Thesis

The thesis puts forward a concept of a *distributed persistent object model* for the programming of persistent object-oriented distributed systems in CORBA.

The *distributed persistent object model*, presented in Section 4.2, is an execution model for distributed persistent object-oriented applications. The principal issue of this work was the development of a model that, on the one hand, provides *the design and the implementation* of persistent CORBA objects and, on the other hand, allows a *soft, non-intrusive integration* of existing databases. Thus, the model has the following distinct properties.

- *Transparent persistence.* The model extends the OMG model with concepts for realising distributed object-oriented communication with persistent database management systems. Binding together distributed (OMG) and database data models (ODMG), the model provides a single paradigm for designing distributed persistent objects. Thus, the basic advantage of the model is a property of *transparent persistence*. There is no difference in application client code working with either distributed transient or persistent objects. The model transparently allows the traversal and modification of the persistent data structures without explicit calls to read and write the data. Moreover, a remote client cannot only read persistent data, but also modify and commit the modifications so that their effects are permanently recorded in persistent storage.

The property of *transparent persistence* can be very attractive for „*pure*“ CORBA clients, which do not know anything about the persistence of processed objects. The model hides the overhead imposed by the persistence and makes this feature transparent. However, it also provides efficient mechanisms for *automatic transaction control* and *management of server objects*. Moreover, for advanced clients the model also gives the possibility of *customising the behaviour* of these mechanisms and *access to the native database*

functionality, i.e. full control of the management of the databases, transactions and the clustering of persistent objects.

- *A soft, non-intrusive integration.* Since a schema evolution may be a very time-consuming process for large and complex data volumes, realising a *mediator-based approach* allows the model to achieve the CORBA-conformance without any modifications to existing databases and applications. It is also important that databases that are remotely accessible via CORBA are at the same time available for native database applications.

On the one hand, the introduction of intermediate mediator components enhances the *flexibility* and *adaptability* of the integrated system, as well as allows the realisation of „*object views*“ of the structure of persistent objects. On the other hand, however, it brings a burden for mediator management. Our own contribution here consists in identifying the important components of the programming model for *robust mediator management*, pointing out design alternatives, and collecting those components into a well-rounded execution model.

Based on this model, the thesis then proceeds to discuss the design and implementation of the *eXtensible Database Adapter (XDA)*. Implementing the proposed model, the XDA is unique in that it permits object-oriented programming of the CORBA interfaces to existing ODMG-compatible database management systems. XDA is a running prototype system developed as part of this thesis. At the time of writing, the XDA supports the ObjectStore database management system [ODIug01].

An additional contribution is also to demonstrate the vitality of the proposed concept by way of concrete implementations. Actually, two XDA prototypes were developed. The first version allowed us to identify the *crucial problems* appearing in the integration process and to propose *corresponding solutions*, which were realised in the second version. They are described in Chapter 6. The prototype developed was used and evaluated in several research projects [BCG+98, BCG+00, SS01, STCK02]. Results for one of these projects are presented in Chapter 7. The main distinctive features of the XDA are as follows.

- **Easy implementation of efficient and adaptable mediators.** The proposed framework *simplifies* the development of interconnecting mediator components. The framework not only gives the necessary *guidelines* for building *scalable well-performing* CORBA/ODBMS applications, but also reduces the costs of their development, providing *adaptable pre-defined ready to use components* (Section 6.2). Some additional techniques, such as „*bulk*“ *operations*, described in Section 6.5, can be incorporated into the code. Two alternative approaches intended to simplify the development of ODBMS adapters are studied: *dynamic mediators* and *automated code generation* (Sections 6.2.2 and 6.2.3). Due to the flexible design, XDA can be configured for various ODMG-compatible database management systems.
- **An efficient and flexible ODA core.** New approaches for efficient dispatching and transaction management are proposed. Providing transparency of persistent objects, XDA realises efficient automatic management of database-specific functionalities, such as transactions and clustering (Section 6.3). At the same time, for advanced clients the adapter provides the possibility of *customising the behaviour* of these mechanisms and accessing the *native database functionality*, e.g. manual control of databases, transactions and clustering of persistent objects.

Further, to improve the *performance* and *scalability* of the integrated system on databases with large numbers of persistent objects the XDA performs *efficient dispatching* of requests to persistent objects and *intelligent caching* of corresponding

mediators. A new approach for efficient persistence-aware request dispatching is presented in the Section 6.4.

The immediate goal of developing XDA is to provide high-level connectivity to object-oriented databases for multiple heterogeneous clients. A long-term objective is to provide an experimental framework for research on the resolution of semantic heterogeneity.

Structure of the Text

The remainder of the thesis is organised as follows.

Chapter 2 first discusses the background material on CORBA and ODBMSs, then the *reasons for integrating* both technologies. The basic issues concerning the integration are highlighted in Chapter 3. In addition, the *evolution* of CORBA/ODBMS integration tools is presented and discussed. Chapter 4 presents the main design issues of our approach. A description of the basic methodology of the *distributed persistent object model* in the earlier sections of this chapter is followed by the design of the supporting XDA framework and its major components.

The main implementation issues concerning the components of the XDA prototype are described in Chapter 5. The first prototype of the XDA has been designed and implemented to verify our model for CORBA/Database integration and to confirm its viability. In fact, Section 5.6 presents *crucial problems*, which have been identified within some research projects using the prototype of an XDA-based *Spatial Object Database Adapter* (SODA) to access real geospatial databases.

Most importantly, Chapter 6 presents the *corresponding solutions* to the problems shown in Chapter 5. The first Section 6.1 describes the new POA-based architecture of the adapter. Then, in Section 6.2, a new method for the implementation of *complex mediators* („views“) is proposed. Two alternative approaches intended to simplify development of the ODBMS-adapters are studied: *dynamic mediators* and *automated code generation*. The following Sections 6.2.2 and 6.2.3 propose new algorithms for *efficient request dispatching* and *transaction management*, respectively. Experimental results are presented to confirm the efficiency of the proposed methods. Finally Section 6.5 proposes various methods aimed at boosting the performance of the resulting CORBA/ODBMS system.

Results of a *practical evaluation* of the XDA are presented in Chapter 7. Here we discuss XDA usage for the development of CORBA integration for *GeoToolKit*, a spatial object-oriented database developed at the University of Bonn [BBC97]. A short but informative *case study* that demonstrates the way in which XDA can be deployed in the practice is given in the Section 7.2. The last Section 7.3 *evaluates the performance* of the CORBA/ODBMS applications which have been developed and shows the *differences* between XDA-based and native database applications.

Finally, Chapter 8 summarises the thesis and discusses some possible future developments in both technologies and their potential influence on the issues of the CORBA/ODBMS integration.

Chapter 2

Basics of Object-Oriented Databases and CORBA

This chapter presents a review of CORBA and object-oriented database management systems (ODBMS). It describes technologies and outlines motivations for their integration.

First, the fundamentals and basic ideas of the ODBMS and CORBA technologies are presented. Then the advantages and drawbacks of both technologies are discussed. Finally, the potential benefits of the CORBA/ODBMS integration are estimated, classified and evaluated.

2.1 Object-Oriented Database Management Systems

Researchers became interested in ODBMSs in the late 1970s and early 1980s as object-oriented systems started to appear. In [ABD+89] Atkinson and others gave a guideline on the requirements for an ODBMS, and this document is still considered the major manifesto for ODBMS. They use the concept of an object, which is a collection of data items and operations, which can access and modify corresponding data items. This research was also done to overcome many restrictions imposed by the relational model on certain types of data [ABC+93]. This included data for organizations such as CERN, the medical profession and the telecommunications industry. All these groups have large amounts of data, which needs to be stored and manipulated in ways which relational systems were not designed to handle. These limitations are starting to become less visible, as many relational systems are starting to add object-oriented functions and interfaces in the hybrid object relational model.

The first commercially available ODBMSs became available in the mid-1980s. By the early 1990s there were a range of ODBMSs available from a variety of vendors. Today object-oriented databases are used for various tasks and mission-critical applications (CAD/CAE/CASE, finances, telecommunication etc.). They are still a specialised field, however, they continue to gain market acceptance as object oriented methods become more common [Heu92].

Modern ODBMSs come in different designs, which determine how objects are transferred, maintained, and controlled [KM94]. This chapter presents the most important conceptual designs and performance issues of ODBMSs.

2.1.1 ODMG Standard

First ODBMSs differ on several issues, for example multiple versus single inheritance, and either the operations/methods or only the data is stored. The sources of these differences are caused particularly by different capabilities of object-oriented programming languages (or their combination), which ODBMSs use. For example, C++ supports multiple inheritance, so all ODBMSs that support C++ as the database programming language must support multiple inheritance. If an ODBMS supports different programming languages, it probably should store in a database only the interfaces rather than the implementations of the methods. With these problems, it is difficult to reach an agreement on all parts of the modelling perspective [LV97].

Therefore, the first ODBMSs lacked an universally accepted data-model such as RDBMS and some researches considered this a major weakness of ODBMSs [DD95, SRL+90], while

others looked on it from a different perspective – ODBMSs are new and only research into the area will give the answer [Bee90, Mai89]. The lack of a commonly agreed data model and definition language was argued to be one of the biggest problems associated with the object-oriented data model [DD95]. It was a problem, but whether it was a conceptual or an implementation problem remain to be discussed. Even now, there is no consensus on a common object data model, but several proposals exist, for example [CB99 and SR92].

To close this gap, in 1993 the Object Database Management Group (ODMG) put forward the first ODMG-93 standard [Cat94], that followed in 1997 by the ODMG 2.0 [CB97] and in 1999 the ODMG 3.0 standard was released [CB99], which is today the „de facto“ standard for ODBMSs.

The main goal of the ODMG standard is the portability of the database source code. In other words, it enables standard-compliant database applications to be ported without changes to any standard-compliant DBMS¹. The ODMG philosophy is very close to the OMG's² and defines an *object model*, an *object definition language* (ODL), an *object query language* (OQL) and interfaces for the programming languages C⁺⁺, Smalltalk and Java.

Object Model. The object model gives a formal meaning to the basic concepts determining how information is structured in a database. One important goal of the ODMG object model is to avoid an impedance mismatch³, i.e. a different representations of data in the ODBMS and in the object-oriented programming languages supported by the ODMG [CM84].

As a whole, the ODMG object model is close to the C⁺⁺ object model. In particular, the model introduces the following core concepts: persistent objects, attributes, relationships, queries, transactions and specifies the basic modelling primitives (object and literal), types (by which objects are categorized), object properties (attributes and relationships) and operations on objects. The transactions and other basic properties of databases are also defined. In addition, since release 2.0 a *meta model* (database schema) is introduced.

The ODMG Object Model distinguishes the inheritance of state from the inheritance of behaviour and expresses the notion of extents and keys. It also defines basic properties and operations on objects stored in a database. In addition, the object model defines standard interfaces for different types of collection objects, iterators, structured objects (for example date, time) and exceptions.

Object Definition Language (ODL). The ODL is a unified language used for the database schema definition. The ODL syntax is a direct superset of the IDL syntax. New keywords supporting relationships, extents, keys, collection, and structured types are introduced. The ODL is independent of the programming language (C⁺⁺, Java, Smalltalk). A pre-processor translates the schema into class definitions for the programming language.

¹ In contrast to CORBA, which is an *interoperability* standard (allows standard-compliant applications to interoperate with standard-compliant systems), the ODMG is a *portability* standard (allows standard compliant applications to be ported between standard-compliant systems).

² In fact, initially ODMG was operating as a subgroup of OMG called Special Interest Group on Object-Oriented Databases (SIGODB).

³ When an application has a data/object-model that is different from the database's data model, an *impedance mismatch* exists. The difference makes it necessary to have some kind of mapping between the models. The extra code increases the complexity of the system, introduces new sources for errors, and puts a number of restrictions on further development of the system. It may be large (e.g. as in RDBMSs), small (e.g. as in ODBMSs) or non-existent (e.g. as in orthogonally persistent programming languages). The result is a complex model with about 30% extra code added [King78] due to the impedance mismatch.

Object Query Language (OQL). The ODMG standard defines a declarative (non-procedural) language, which can be used to query and update database objects. The OQL is based on the O₂SQL (which in turn is based on SQL92). In addition to SQL-style queries, OQL can be used to query complex types of data; it can make use of the methods in classes; it can be applied on all collection types mentioned above. The OQL does not support recursive queries.

Language bindings. Language bindings map concepts of the object model (expressed via ODL) to the major object-oriented programming languages like C++, Java and Smalltalk. Language bindings define language-specific APIs through which programmers can manipulate objects stored in a database. For example, the C++ binding is defined by means of specific libraries, which provide classes and functions implementing the concepts defined in the object model. Unfortunately, only a subset of the object model and of the OQL key-words can be used without restrictions in ODMG-compliant programming code (such as C++). Therefore, the actual implementation of the language binding is still highly vendor dependent.

To be *ODMG-compliant*, a database system must conform to only one of the following four points: ODL, OQL, Language binding. According to this weak definition, almost all members of the ODMG can provide ODMG-compliant database systems⁴. The conditions to call a database system *ODMG-certified* are more strict: the database system must conform to *all* ODMG criteria; this is to be verified by official tests of the ODMG. However, not a single database system has yet been ODMG-certified! For this reason the term *ODMG-standard* is slightly misleading – it is rather a guideline for a future standard. Currently, individual parts of the standard are supported by particular ODBMS systems.

2.1.2 ObjectStore as an Example of ODBMS

As an implementation of an ODBMS we decided to choose ObjectStore from Object Design Inc. [LLOW91, ODIug01]. It is one of the most sophisticated and popular commercial object-oriented databases available on the market [Dick97]. Besides traditional C++ language binding, the current version of ObjectStore (6.0) supports Java and ActiveX. To describe ObjectStore briefly, let us refer to the general issues discussed in the previous sections.

2.1.2.1 Object Model

The lifetime of regular objects is either *coterminous with a procedure* (objects being function arguments and automatic variables) or *coterminous with a process* (static and heap-allocated objects). ODBMSs introduce yet another kind of objects, which lifetime is *coterminous with a database*. Such objects will be further referred to as *persistent objects*, whereas regular objects will be referred to as *transient objects*.

Persistent and transient objects can coexist in the same process. The *lifetime status* (transient or persistent) of particular objects is indicated by programmers. There are several approaches how to employ the standard language constructs to express object persistence. Different ODBMS products can support individual strategies or a mix of them [Khos93]. In ObjectStore the *persistence-by-instantiation* strategy was used. According to this approach, an object is made persistent and gets its persistent capabilities when instantiated. For this purpose ObjectStore provides an overloaded `new ()` operator that creates persistent objects.

⁴ In fact, the standard is criticized by many analysts as being inconsistent, full of discrepancies and confusions [Alag97, Rupp99]. Still, partially due to the lack of meaningful alternatives, the ODMG standard is kept evolving and is supported by virtually all ODBMS vendors.

This strategy is very flexible since instances of any class can be used both persistently and transiently. Persistent instances are automatically and transparently stored by the ODBMS; their values can be used again later. Transient instances only exist while the application based on the ODBMS is running. Methods and other operations can be applied to both persistent and transient instances in the same way.

The basic disadvantage of this method is that once an object is created, its lifetime status cannot be changed. Moreover, since making an object persistent is done explicitly, this strategy does not support *orthogonal persistence*⁵. Another problem connected to this strategy is what happens if a persistent object references a transient one. This may destroy referential integrity and often will be solved by using inverse relationships. By using this kind of relationships, the design of classes and the implementation is made more rigid. Thus, ObjectStore object model offering object-persistence will still retain some of the impedance mismatch.

2.1.2.2 Transactions

Most ODBMSs provide advanced database-specific facilities like transactions processing, concurrency control and crash recovery. These mechanisms are closely connected and interact with each other. The key notion here are transactions, since transaction management is handled explicitly by programmers.

Transactions enclose one or more operations (which normally manipulate persistent data) between *transaction boundaries* determined by commands of *transaction start* and *transaction end*. In formal terms, the transaction can be described as a related set of operations, which have four basis properties, known as ACID properties.

Atomicity. The operations enclosed by transaction boundaries are considered as a unit. Either all the operations complete (the transaction *commits*), or all the operations are undone (the transaction *aborts*).

Consistency. The transaction transforms data from one consistent state to another in predictable manner.

Isolation. The partially updates states of data are not visible outside the transaction itself.

Durability. The outcome of transaction is not reversed (partially or completely) once transaction is completed.

ObjectStore provides a special class `os_transaction` defining static operations and types needed for manipulation with transactions. In particular, the `begin` method takes an argument of enumerated type denoting the type of transaction to be started (`read_only` or `update`).

Like most database management systems, ObjectStore tries to interleave the operations of different processes' transactions to increase concurrent usage of resources. When scheduling the operations, ObjectStore conforms to a strict two-phase locking protocol, except in the case of multiversion concurrency control (MVCC). This protocol has been proven correct in that it guarantees serialisability. That is, it guarantees that the results of the scheduling are the same as the results of some noninterleaved schedule of the transactions' operations.

⁵ In other words, the program should not be changed to offer persistence of its objects, resulting in a total removal of the difference in holding short-term data (exists only while a program is running) or long-term data (outlives the program, on file or in database). Several formal definitions of orthogonal persistence have been proposed, for example [ABC+93 and AM95].

2.1.2.3 Containers and Queries

ObjectStore provides a hierarchy of collection template classes similar to the one described in [CB99]. On the top of the hierarchy is the `os_Collection<T>` class, which defines basic operations for the manipulation with elements (`insert`, `delete`, `retrieve`), typical set operations (`join`, `union`, `intersection`) and operations controlling the behaviour and representation of the collection. ObjectStore enables programmers to choose between unordered and ordered collections that either do or do not allow duplicates. Each class derived from the root collection class (`os_Bag<T>`, `os_Set<T>`, `os_List<T>`, and `os_Array<T>`) implements a different strategy of elements grouping. One important limit of ObjectStore collections is that *the collections' element type must be a pointer type*. To support navigation within a collection, ObjectStore provides an iterator template class called `os_Cursor<T>`.

Among the database services provided by ObjectStore is the support for query processing. The queries are used from within C++ programs. They treat persistent and nonpersistent data uniformly. The ObjectStore collection classes contain a query method through which specific collection members can be selected. The query method takes a *query string* as an input, which is a valid C++ *control expression* indicating the selection criterion of the query. The control expression has to evaluate to an integer. As a result, the query method returns a heap-allocated collection of elements. Query strings typically contain boolean expressions comparing data members of objects being the collection's elements with some constant values. Functions called in query strings are subject to certain limits [ODIcg01]. ObjectStore parses the query string at runtime and checks the syntax and semantics before execution. If there is an error in query string, ObjectStore raises an exception. For preanalysed queries, the query string can also include calls to other nonoverloaded global functions as long as some restrictions are met. Variables (including data members) can appear in a query string as long as the type of each variable (except data members) is specified explicitly with a cast.

2.1.2.4 Database References

The issue of standard C++ pointers and references use has special importance for C++-based ODBMSs. Since persistent and transient objects are treated by the application equally, they can hold pointers/references to each other. However, when the application process terminates, the connections between transient and persistent data structures will become invalid and the data integrity will be violated⁶. Therefore, each ODBMS places some restrictions on the use of C++ pointer and reference types linking persistent and transient objects. The common restriction is that C++ pointers and references are meaningful only during the execution of the transaction. They are invalidated when the enclosing transaction commits. It is right not only for references from persistent to transient memory, but also for references from transient to persistent memory.

To solve the problem, most ODBMSs support *database references*, or in other words, references to persistent objects. Every database reference reflects the identity of the persistent object it refers. In other words, database references are the C++ counterparts of object identifiers (Object IDs) through which the concept of object identity is realised in Object-Oriented Databases. Database references always preserve their validity.

The standard C++ pointers in ObjectStore applications are subject to the same restrictions described above. Pointers from transient memory to persistent memory and pointers from persistent memory to transient memory are valid until the end of the current transaction,

⁶ ...unless ODBMS supports *persistence-by-reachability* strategy. Still, that strategy is most typical for ODBMSs based on languages with incorporated garbage collection (such as Smalltalk or Java).

whereas pointers from transient memory to transient memory and pointers from persistent memory to persistent memory are always valid.

2.1.2.5 Entry Points

Entry points are well-defined objects stored in the database, which are associated with human-declared names. The ODBMS provides a simple operation taking the entry point name and returning pointer/reference to the associated entry point object. All other objects are accessed through the navigation (following pointers) beginning from the entry point objects.

ObjectStore's equivalents of entry points are *database roots*. A database root is a persistent entity that is stored in the database. The sole purpose of database roots is to enable the association of arbitrary names with persistent user-defined objects. After the association has been established, the current process or any other process can retrieve the associated object and use it as the entry point to navigate to other objects in the database.

Database roots are implemented through a special `os_database_root` class. The `os_database_root` is a simple class containing a pair of variables (a pointer to root object and an associated name string), and defining operations for getting/setting the values of those variables. The operations for creation and retrieval of roots are accessible through a static class `os_database` that also defines global database-specific operations like creating/destroying or opening/closing databases.

2.1.2.6 Allocation Units

The following Figure 1 illustrates the hierarchical relationship among all allocation units in ObjectStore: pages, clusters, segments, databases and file systems. The basic unit of allocation in an ObjectStore database is the *cluster*. Clustering is a common optimisation for reducing the number of disk transfers needed to access persistently stored objects. The technique is to allocate contiguous (or nearly contiguous) storage for objects that are accessed together frequently. When designing an application with clustering in mind, the programmer asks: „If my application accesses this object, what other objects is it likely to access along with it?“ Optimal clustering can be achieved by allocating related objects in the same cluster.

Clusters are organised logically into segments. If the cluster is the level for grouping objects for the best performance, the segment is the level for grouping clusters that are logically related. That is, segments allow to group clusters (and the objects they contain) according to certain shared characteristics. Typically, the clusters are used to control data locality and segments to set the logical characteristics (for example, access control) of a group of clusters.

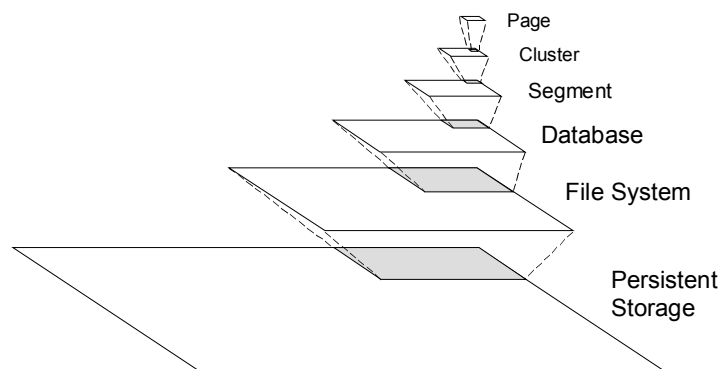


Figure 1. The hierarchical relationship among allocation units in ObjectStore

2.1.2.7 Client-Server Architecture

Almost all ODBMSs are based on client/server architecture. One or more *database servers* manage their local storage devices and provide networked *database clients* with access to objects persistently stored in the database [GUW02, SKS97].

The most C⁺⁺-based ODBMS servers transfer objects referenced by the client to the client side, where their operations are executed. In fact, the objects' *states* are stored in the database⁷ and transferred to clients, whereas the objects' *operations* are provided by client applications. A notable exception here is the ODBMS system O₂ [Mano94], which supports server-side object execution. In general, however, most C⁺⁺-based ODBMSs execute objects' methods exclusively on the client side.

With respect to the granularity of data transfer between the server and the client, ODBMSs can be divided between *page servers* (clients and servers interact in terms of pages) and *object servers* (clients and servers interact in terms of objects). Each strategy has its strong and weak points. See Table 1 for their brief comparison⁸.

	Object server	Page server
Transfer unit between client and server	<ul style="list-style-type: none"> Object 	<ul style="list-style-type: none"> Page
Tasks performed by client	<ul style="list-style-type: none"> object cache management 	<ul style="list-style-type: none"> index management page cache management
Tasks performed by server	<ul style="list-style-type: none"> object cache management log/lock management index management page cache management storage allocation 	<ul style="list-style-type: none"> page cache management log/lock management storage allocation
Strong points	<ul style="list-style-type: none"> better concurrency control (object-level locking) queries can be executed either on server or on client 	<ul style="list-style-type: none"> processing is distributed between clients and servers; servers are not overloaded better performance on small interconnected objects⁹

Table 1. Comparison of page and object servers

When data is transferred between clients and servers, some additional information is needed to define how to interpret the stream of bytes, to identify and extract the logical objects. The ODBMS must be able to determine the exact boundaries and the layout of objects' internal structures. Typically, since C⁺⁺ classes are not runtime entities, each C⁺⁺-based ODBMS keeps the so-called *schema information* describing the layout of classes whose instances are stored in the database. In addition, every client application willing to store/retrieve data in the ODBMS must provide its own schema information describing the layout of classes of persistent objects used by this application. When a new class's instantiation (object) is created and stored in the database, its layout information is added to the database's schema information. This information is further used for runtime type checking. Thus, when

⁷ Therefore, some analysts blame ODBMSs for being object *state* databases, not real object-oriented databases.

⁸ It should be noted, however, that the majority of C⁺⁺-based ODBMSs are page servers.

⁹ Most page servers introduce facilities for *object clustering* – placing interconnected objects on the same page.

(possibly another) application tries to store/retrieve an object of the same class, the layout of the class used in the application is checked (at runtime) against the layout of the relevant class stored in the database to make sure, they match each other.

ObjectStore is based on the client/server paradigm and requires two auxiliary processes for application execution: an ObjectStore *server* and a *cache manager* (Figure 2) [ODIba01]. The non-networked ObjectStore/Single application provides the same functions as the networked ObjectStore, but the two processes – server and cache manager – have been combined into one process.

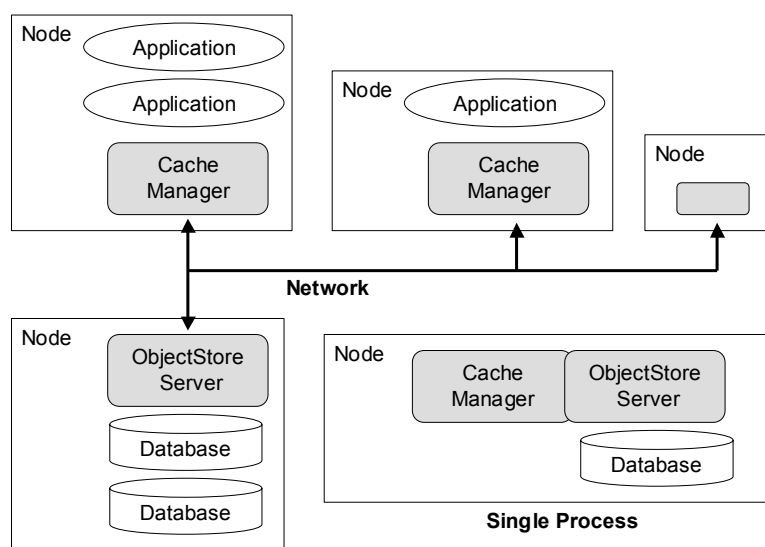


Figure 2. Schema of the typical networked ObjectStore application

The *cache manager* is a UNIX daemon (a Windows service) that runs on the same machine as the client application. In contrast to the server, the *cache manager* is started automatically when an ObjectStore application starts. If additional ObjectStore applications are started on the same machine, they all use the same cache manager. Its main role is to respond to server requests as a stand-in for the client application and manage the application’s client cache, which plays the role of a local holding area for data loaded from the server.

The *server* handles access to the ObjectStore databases, including the storage and retrieval of persistent data. The system administrator typically arranges for each server to start when its host machine boots.

ObjectStore has a typical architecture of a page server. It transfers data (pages) between database memory (persistent) and program memory (transient) automatically and transparently to the user. ObjectStore’s *virtual memory mapping architecture* (VMMA) uses page faulting to detect a reference in a running program to persistent data. When a page fault occurs, ObjectStore transfers the page containing the referenced data automatically – possibly together with adjacent pages – across the network to the application’s cache. The page containing the referenced data is then mapped into virtual memory. After persistent data has been mapped into virtual memory, access to it is as fast as access to transient data.

The granularity of server transfers can be changed from a page to a segment, with the latter resulting in transfer of a segment (all pages belonging to the segment). For each transaction, only those parts of the database(s) that are accessed by the transaction are mapped into the address space of the client. This strategy introduces a restriction on the total amount of data that can be referred to by any single transaction. Large operations need to be broken down into a series of smaller transactions.

When a page is mapped into the virtual address space, ObjectStore dynamically assigns a virtual address where the mapping is to take place. An attempt is made to assign the address so that the pointers stored on the server continue to be valid in virtual memory of the client, which is possible when the page being mapped as well as all the pages referred to by pointers embedded in the mapped page can be assigned the same virtual addresses where the pages were last resident. No pointers need to be swizzled in this case and execution can continue as soon as the page is mapped. In all other cases, the server has to find all the pointers embedded into the page and swizzle the pointers as needed, which requires that some portion of the type system be available at runtime in order to find all embedded pointers. ObjectStore keeps this information in an auxiliary data structure called the tag table, which records the location and type of every object in the database. The tag table is used in conjunction with the database schema to find all pointers embedded in objects stored in the page being mapped.

At the end of a transaction, all pages in the client's address space are unmapped and any modified pages are transmitted back to the server; the client blocks until the pages are written back to the server's disk(s). Unmapped pages stay in the client's cache until room is needed for other new pages. A client cache coherency scheme is used to accommodate sharing of pages by multiple clients.

2.1.2.8 Multiple Language Access

Most ODBMSs (especially page servers) use a *memory image* approach to store objects in databases. The memory representation of persistent objects follows closely the format used by the programming language of the ODBMS. Thus, the overhead of conversions needed during the objects' move between persistent store and transient memory, is reduced. This significantly improves performance of the DBMS. However, this strategy is a two-edge sword: improving the performance, it ties a DBMS to a particular programming language (in our case C⁺⁺). Access to persistent objects from other programming languages is severely limited, because all data structures from one language should be mapped into data structures of another language.¹⁰ Usually full database functionality is provided only for a primary language (the one to which the ODBMS adds database functionality) allowing only a limited functionality for other languages.

The current version of ObjectStore supports only C⁺⁺ and Java programming languages.

2.1.2.9 Building ObjectStore Applications

To realise the advantages inherent in its memory mapping architecture, the ObjectStore needs to store information in each database about the classes of objects stored and about the layout of instances of these classes. This schema information allows ObjectStore to identify the locations of pointer fields in each newly retrieved cluster, so it can perform relocation. Thus, an ObjectStore client application consists of two parts: an *executable* and a *schema database*. In the schema database ObjectStore stores schema information in form of C⁺⁺ objects. Classes themselves are not runtime objects in C⁺⁺. Therefore, ObjectStore must create representations of classes in compile time.

2.1.2.10 The Metaobject Protocol

Yet another important feature of ObjectStore – the Metaobject Protocol (MOP) deserves to be mentioned. MOP is a library of classes that allows accessing the ObjectStore schema information at runtime. Schema information for ObjectStore databases and applications is stored in the form of objects that represent C⁺⁺ types – *metaobjects*. These objects are actually

¹⁰ The set of supported programming languages is usually limited to C⁺⁺, Java and Smalltalk.

instances of the ObjectStore *metatypes*, so-called because they are types whose instances represent types. Every *metaobject* representing a type is an instance of a subtype of the metatype `os_type`, which defines basic operations characteristic for all types¹¹. MOP has also some other classes that instances represent schema objects other than types, such as `os_base_class` and `os_member` and their subtypes. These auxiliary classes are part of the MOP, but they are not the *metatypes* and, therefore, do not included in the metatype hierarchy.

2.2 The Common Object Request Broker Architecture

The *Common Object Request Broker Architecture* (CORBA) standard is specified and maintained by the Object Management Group (OMG) – a consortium associating all major software and hardware vendors as well as various research institutions and chief end-users. The key objective of OMG is to create a component-based software marketplace by hastening the introduction of standardised distributed object software. In particular, the first key specifications produced by the OMG – the *Object Management Architecture* (OMA) and its core, the CORBA specification – provide an architectural framework that is both rich and flexible enough to accommodate a wide variety of distributed systems (Figure 3).

Like all technologies, CORBA has a unique terminology associated with it. Although some of the concepts and terms are borrowed from similar technologies, others are new or different. Understanding these terms and the concepts behind them is important for the following discussion. After describing the most important terms of OMA and CORBA¹², the remainder of this chapter provides a high-level review of the computing model, the architecture, and the important concepts of CORBA.

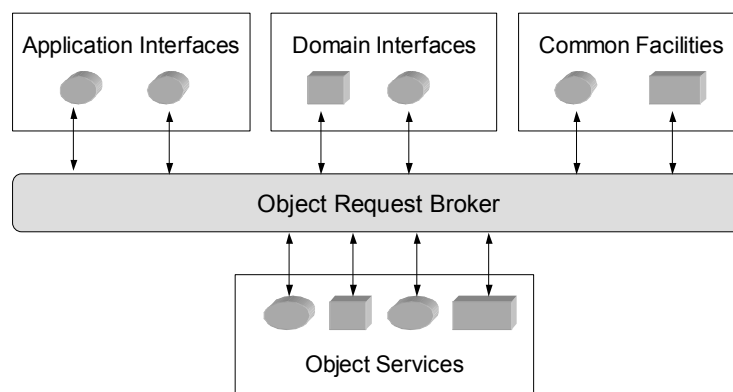


Figure 3. The Object Management Architecture

2.2.1 Why CORBA?

CORBA was introduced to standardise the development and deployment of applications operating in distributed heterogeneous environments. The main idea is to standardise how distributed heterogeneous clients and servers can interoperate in terms of high-level object-oriented programming languages. Using an ORB, new software can easily be added to the system (plugged into the bus). Further, old legacy software can be integrated into newer systems in a flexible way because of the language independence. The CORBA standard was developed with the component based software development model in mind. Software is

¹¹ For more detailed information about the hierarchy look at [ODIaug01].

¹² A full description of CORBA is beyond the scope of this thesis. Instead, some aspects of CORBA relevant to the CORBA/ODBMS integration are discussed. For more detailed information about CORBA see the specification of the standard [OMGorb01], OMG's web site <http://www.omg.org/>, or for less formal introduction to CORBA, see [OHE96].

developed in components or packages with predefined external interfaces. Hiding the implementation details, these components should be easy to combine and „plug in“ anywhere to get the desired functionality. A distributed CORBA object is a component in that sense. By developing components with clear defined interfaces, the historically expensive integration phase of software development can be eliminated [Baker97].

2.2.2 The Object Management Architecture

The ORB fits into a higher-level architecture defined by OMG called the Object Management Architecture (OMA) [OMGoma00] shown in Figure 3. Four different kinds of CORBA objects can be plugged into the ORB: application specific objects, standardised domain objects (e.g. for the medical domain), standardised common facility objects (e.g. system management), and finally the generic CORBA services [OMGserv96] that can be deployed in any CORBA system (e.g. Transactions, Events). All CORBA objects must be defined with external interfaces in order to be connected to the ORB [OMGorb01].

2.2.3 Object Request Broker

Figure 4 shows the conceptual architecture of an ORB and relationships between most of those CORBA components, which we describe in the following sections. The other elements of the CORBA architecture are not directly related with the subject of this work and therefore are excluded from this discussion.

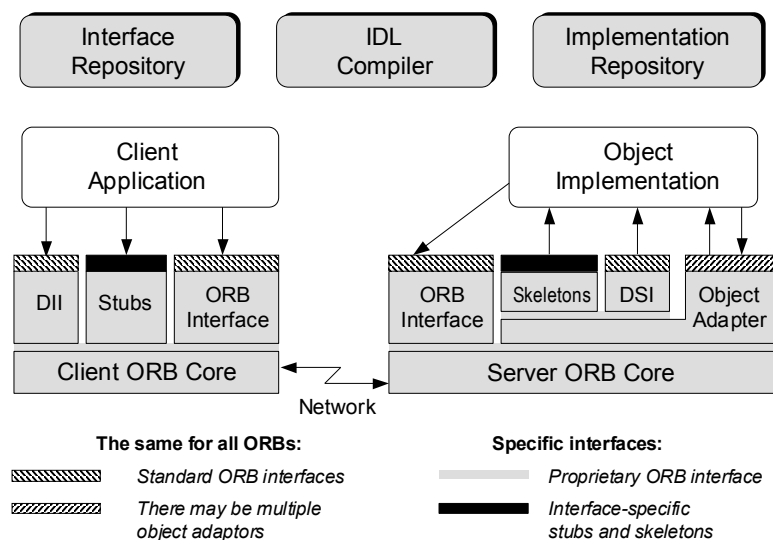


Figure 4. The main components of the CORBA

A *CORBA object* is a „virtual“ entity capable of being located by an ORB and having client requests invoked on it. It is virtual in the sense that it does not really exist unless it is made concrete by an *object implementation* written in a programming language. In CORBA, the *object* exists only as an abstraction. It is visible through operations defined by an interface and invoked using an *object reference*. The object reference itself is an abstraction, and is available to clients only as an opaque value, which can be passed as a parameter in an operation, or stringified to give an external representation that can be stored and subsequently turned back into an object reference.

A *server* is a computational context in which the implementation of an object exists. A *client* is a computational context that makes requests on an object through one of its object references. The roles of the both, a *server* and a *client*, are depend on a given communication

context. For example, a program that is the *client* for one object may be the *server* for another. The same process may play both, *client* and *server* roles for a single object.

Object Request Broker (ORB) is the central component of OMA. The *ORB core* takes care of low-level operations characteristic for distributed computing and acts as object bus over which objects transparently interact with each other. Thus, the core hides from the objects the location, communication medium, implementation details, and execution state of the process where service is actually executed. The main responsibilities of the ORB core are related with transparent transport of service requests and responses.

The *ORB Interface* provides general functionality such as the conversion of objects to strings and vice versa as well as access to the *interface* and *implementation repositories*. This interface is identical for all implementations and can be accessed by clients as well as servers. Either way, the client directs the request into the ORB core linked into its process.

The Interface Repository is a storage place where information about server objects' interfaces is stored. It allows a runtime access to IDL definitions and, therefore, intended for the use in dynamic invocations.

The Implementation Repository is server-side storage, which contains information about an application implementing a particular CORBA server. In fact, the implementation repository serves as a mediator, mapping abstract CORBA servers into real server processes, which fulfil clients' requests.

A *request* is an invocation of an operation on a CORBA object by a client. Requests flow from a client to the target object in the server, and the target object sends the results back in a response, if the request requires one. Thus, requests flow down from the client application, through the ORB, and up into the object implementation in the following manner.

1. The client can choose one of two variants: to make requests using static *stubs (proxies)* compiled into the client's programming language from the object's interface definition or using the *Dynamic Invocation Interface (DII)*. DII interface allows a client to use the underlying request mechanisms provided by an ORB. Applications use DII to issue requests to objects dynamically without needing IDL interface-specific stubs to be linked in.
2. The *client ORB core* sends the request to the ORB core linked with the server application.
3. The *server ORB core* receives the request and forwards it to the *object adapter* that further dispatches the request to the *object implementation (servant)* that is implementing the target object. The *object adapter (OA)* is responsible for providing basic functionality for objects and servers, such as processing object references, activation and deactivation of objects and method invocations. *Object adapters* can be specialised to support certain object implementation styles (such as Object Database Adapters).

Like the client, the server can choose between static and dynamic dispatching mechanisms for its servants. It can rely on static skeletons compiled into the server's programming language from the IDL or use the *Dynamic Skeleton Interface (DSI)*. The DSI is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

4. Finally, the *object implementation* returns the response to the client application.

The next few sections describe the CORBA components required to make requests and to get responses.

2.2.4 CORBA Application Development Steps

Figure 5 shows a typical scenario of CORBA application development. In this case the client and the server use different programming languages – Java and C++, respectively. The client and server developers are completely independent from each other and use different development environments, programming languages and CORBA implementations. The only link between them is the IDL interface definition (1) which the both use. As long as both applications support the same IDL interface, client and server can be modified arbitrarily without need of recompilation of the counterpart.

An IDL compiler translates the language-independent definition (1) and produces platform-specific source files that must be combined with application code to produce client and server executables. The skeletons (2) and the server implementation source code (3) are compiled and linked into the server executable (4). Similarly, the stub source code (5) and the client implementation source code (6) are compiled and linked into the client executable (7). Both client and server also link with an ORB library that provides the necessary run-time support.

It should be noted that only a conceptual view of this process can be presented here, because CORBA does not standardise the development environment. This means that details, such as the names and number of created source files, vary from ORB to ORB. However, the concepts are the same for all ORB and implementation languages.

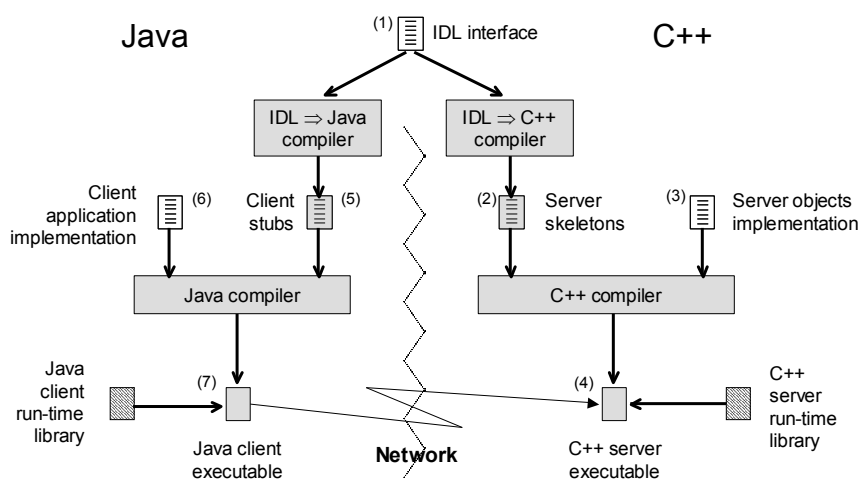


Figure 5. CORBA application development process

2.2.5 The Interface Definition Language (IDL)

To invoke operations on a distributed CORBA object, a client must know the interface offered by the object. An object's *interface* is composed of the operations it supports and the types of data that can be passed to and from those operations. Clients also require knowledge of the purpose and semantics of the operations they want to invoke.

In order to provide a unified platform-independent description of objects' interfaces, the OMG introduced a new language – *Interface Definition Language (IDL)*. The sole purpose of the IDL is to allow object interfaces to be defined in a manner that is independent of any particular programming language like C++ or Java. This arrangement allows applications implemented in different programming languages to interoperate. The language

independence of the IDL is critical to the CORBA goal of supporting heterogeneous systems and the integration of separately developed applications.

The central elements of IDL are *interfaces* defining operations and attributes of server objects. A key feature of IDL interfaces is that they can inherit from one or more other interfaces. This arrangement allows new interfaces to be defined in terms of existing ones, and objects implementing a new derived interface can be substituted where objects supporting the existing base interfaces are expected. IDL interface inheritance is similar to C++ class inheritance with several important differences (in terms of C++).

- all base interfaces are public virtual;
- all operations are public and virtual;
- operations cannot be redefined in derived interfaces.

A few predefined types can be used in interface definitions. IDL supports built-in simple types, such as signed and unsigned integer types, characters, boolean, and strings, as well as constructed types such as enumerated, types, structures, discriminated unions, sequences (one-dimensional vectors), and exceptions. These types are used to define the parameter types and return types for operations, which in turn are defined within interfaces. IDL also provides a „module“ construct used for name scoping purposes. The types are shown in the following tables.

IDL operations can have zero or more arguments and return zero (void) or one return value. Any IDL operation can raise an exception to signal exceptional conditions occurred during operation's invocation. IDL exceptions can have one or more data members of any IDL type. All types can be used for specification of arguments in IDL operations. Arguments to IDL operations must have their directions declared so that the ORB knows whether their values should be sent from the client to target object, vice versa, or both. IDL defines three argument passing modes: *in*, *out*, and *inout*. Arguments passed only from client to server are passed in *in* mode, i.e. are copied from the client to the server at operation invocation. Arguments can also be declared *out* to indicate that, like return values, they are passed from the target object back to the client. The *inout* keyword indicates an argument that is initialised by the client and then sent from the client to the target object; the object can modify the argument value and return the modified value to the client. The following example shows a simple IDL definition.

```
// IDL
interface EmployeeRegistry {
    Employee lookup(in long emp_number);
};
```

Additionally, IDL has an *object reference* type denoting reference to an object. Object reference will be declared by naming the relevant object's interface name. The essential difference between the object reference type and other types is that in the case when an operation argument/return value is of a certain object type, the operation invocation has *pass-by-reference* semantics. The actual values of the arguments/return value are references to object interfaces and not the object themselves¹³. In all other cases, the invocation has *pass-by-value* semantics, i.e. the actual values of the arguments/return values are copied to the buffer passed between client and server.

¹³ The Object by Value specification supporting passing CORBA objects by value (rather than by reference) as parameters in CORBA object operations is adopted and is a part of CORBA 2.5 [OMGorb01].

2.2.5.1 Interoperable Object Reference (IOR)

Object references are the only way for a client to reach target objects. A client cannot communicate unless it holds an object reference. How, then, does a client obtain references (the client must have at least one reference to start with)? Clients can get CORBA references in several ways.

Object retrieval/creation. The object reference is obtained as a result of operation invocation of any other server object in form of a return value or an out argument. In particular, special CORBA interfaces called *object factories* provide operations supporting other objects creation. Typically, the object factory interface contains one or more operations taking some input parameters and returning reference to newly created object.

CORBA service. Clients can lookup object references through well-known services by supplying some key, like object name (Naming service [OMGnam01]) or object properties (Trading service [OMGtrad00]).

Convert from string. Clients can read the stringified object reference from some input source like a file, e-mail or fax, and invoke `string_to_object` operation accessible through the ORB interface. The source string must be previously created by invoking the `object_to_string` ORB operation upon a valid object reference.

To manipulate object references, clients and servers must use operations defined in the `Object` interface inherited by *every* CORBA object. In particular, the copying object must be done through calls to the `duplicate()` operation, whereas the memory allocated in such way must be reclaimed by invoking the `release()` operation. In other words, CORBA object's `duplicate()` operation is called, if new reference to the object is to be created. If the reference is not needed any more, its `release()` operation is called¹⁴. Whether a reference points to the `OBJECT_NIL` – a special object reference denoting no object – can be tested through calls to the `is_nil()` operation. Other operations, such as getting the object's interface or implementation descriptions, are also available through the `Object` interface.

The object reference acts as a handle that uniquely identifies the target object and encapsulates all the information required by the ORB to send the message to the correct destination. Given the transport and location transparency offered by CORBA, there must be some minimum amount of information encapsulated in every IOR.

2.2.5.2 IDL to C++ Language Mapping

IDL Language mappings specify how IDL is translated into different programming languages. For each IDL construct, a language mapping defines which facilities of the programming language are used to make the construct available to applications. For example, in C++, IDL interfaces are mapped to classes, and operations are mapped to member functions of those classes (Table 2). Object references in C++ are mapped to constructs that support the `operator->` function (that is, either a pointer to a class or an object of a class with an overloaded `operator->` member function). Object references in C, on the other hand, are mapped to opaque pointers (of type `void *`), and operations are mapped to C functions that require an opaque object reference as the first parameter. Language mappings also specify how applications use ORB facilities and how server applications implement servants.

¹⁴ In terms of C++, every CORBA object has internal *reference count* set to 1 when object is created. The *duplicate* and *release* operations respectfully increase and decrease the reference count of object. When release operation is called on object whose reference count is 1, the object is disposed.

Interface operations are mapped to C++ virtual methods. Mapping rules for operation's signatures depends on the type of arguments and return values. The general rules are.

- in arguments are passed either by value or by *constant* reference/pointer;
- out and inout arguments are passed either by value or by reference;
- Return values are passed either by value or by pointer.

OMG IDL language mappings exist for several programming languages. As of this writing, the OMG has standardised language mappings for C, C++, Smalltalk, COBOL, Ada, and Java. The existence of multiple OMG IDL language mappings means that developers can implement different portions of a distributed system in different languages. For example, a developer might write a high-throughput server application in C++ for efficiency and write its clients as Java applets so they can be downloaded in the Web. The language independence of CORBA is the key to its value as an integration technology for heterogeneous systems.

IDL Type	C++ Mapping Type
basic types	CORBA object for each, for example boolean – CORBA::Boolean
module	namespace
interface	class
attribute	pair of public overloaded methods with the same name, one for readonly
operation	public virtual method
enum, struct, array	enum, struct, array
union	class
valuetype	class
string	char *, String_var (use string_alloc(), string_free(), string_dup())
sequence	class, behaves like array, operator []
exception	class derived from CORBA::UserException

Table 2. Examples of IDL to C++ language mapping

2.2.6 Operation Invocation and Dispatching

CORBA applications work by receiving requests or by invoking operations on objects. The ORB sends a request to an object whenever a client invokes an operation on the target object. Figure 6 provides a high-level view of the client and server ORB subsystems involved in dispatching a request.

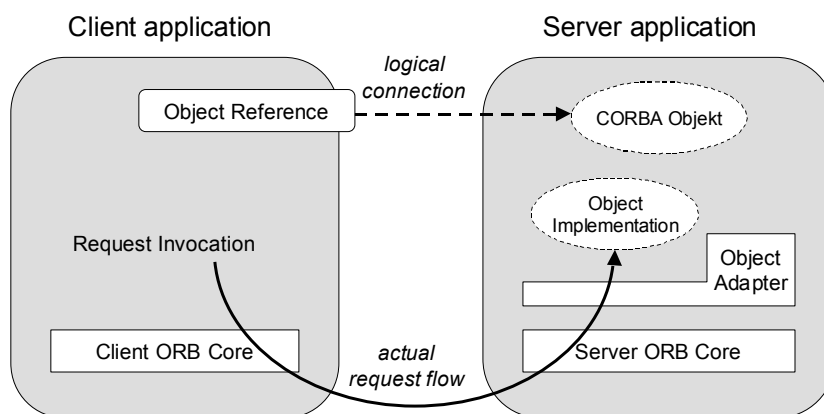


Figure 6. Basic request dispatching in CORBA

First, to send a request to an object, a client must hold an *object reference* and *IDL interface* for the target object. The *object reference* acts as a handle that uniquely identifies the target object and encapsulates all the information required by the ORB to send the message to the correct destination. Therefore, the server application somehow exports an object reference for a CORBA object. The client obtains the exported object reference for the object, perhaps via another request invocation or by receiving it using Naming or Trading Services.

Underneath the client application, the client ORB uses the object reference to determine where the object resides and how to contact it, and then it sends the request to the server ORB. The server ORB receives the request and dispatches it to the *object adapter* hosting the target object, and finally the *object adapter* continues the dispatch by up-calling the implementation of the target object.

Note the distinction between the CORBA object and the object implementation; the CORBA object is a „virtual“ entity that does not really exist unless it will be implemented by an *object implementation*. In Figure 6, the dotted arrow between the object reference and the CORBA object represents the logical connection over which the client ORB sends the request, and the curved arrow shows the actual request flow.

The *object adapters* work as the glue between ORB and object implementations. By applying the adapter pattern [GHJ+94], the object adapter adapts the interface of the object implementation to the interface expected by the ORB. In other words, an object adapter is an interposed object that uses delegation to allow a caller to invoke requests on an object without knowing the object's true interface. The CORBA specification defines two basic approaches for the realisation of this invocation: a *static* and *dynamic*.

2.2.6.1 Static Invocation

The first approach is based on the usage of static code, generated from the IDL interface specifications. They are processed by an IDL compiler, which maps IDL definitions to required client programming language. In the output the IDL compiler generates target programming language code for *stubs* and *skeletons* that cooperate with the ORB to perform the marshalling and unmarshalling on both sides of the client-server connection.

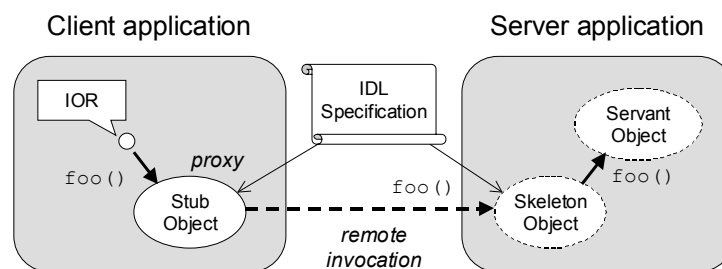


Figure 7. A request invocation

A request invocation with the help of *stub* and *skeleton* runs on the following scenario. When an *object reference* is received by a client, the client-side runtime instantiates a *proxy* object in the client's address space. A *proxy* is an instance of a C++ class from the *stub* that represents the remote *servant object* to the local application. The interface on the *proxy* is the same as the interface on the remote object; when the client invokes an operation on the *proxy*, the *proxy* sends a corresponding message to the remote *servant*. In other words, the proxy delegates requests to the corresponding remote *servant* and acts as a local ambassador for the remote object, as shown in Figure 7. Similarly, a *skeleton* is a server-side function that allows a request invocation received by a server to be dispatched to the appropriate servant. To summarise

the basic principles of the ORB operation, the steps taken by the ORB during static invocation are illustrated in Figure 8. It is assumed, that the target object is already activated.

Skeleton classes do not provide an implementation for IDL interfaces. Instead, they serve as a bridge connecting the *object adapter* with *implementation classes*, which are supplied by the interfaces' programmer. In fact, skeletons express an ORB-expected interface to the target object, whereas the implementation class reflects the interface the actual implementation object provides. To adapt those interfaces one of two approaches defined by classical Adapter pattern [GHJ+94] can be applied.

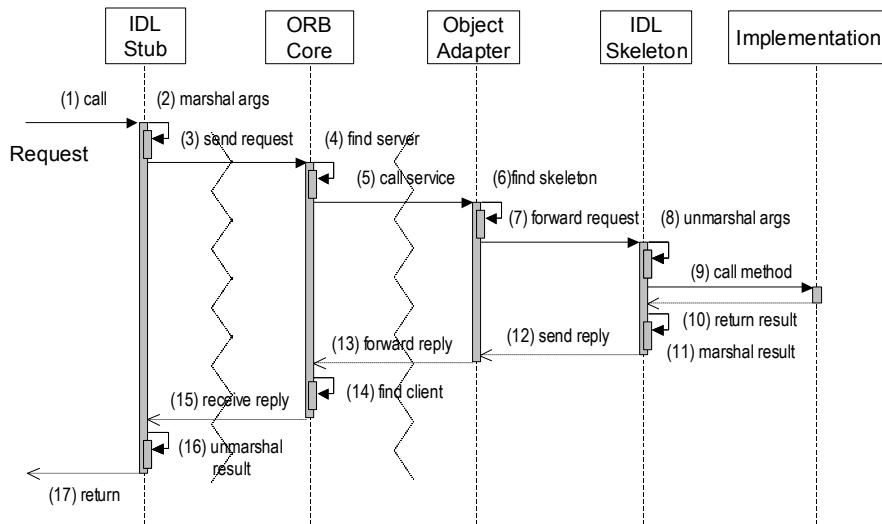


Figure 8. Operation dispatch in static invocation

Inheritance-based Approach

Implementation classes realise IDL operations deriving the signatures of the corresponding operations from the skeleton classes generated by the IDL compiler (Figure 9). Skeleton classes declare pure virtual member functions for every IDL operation defined in the interface. These member functions are redefined by the implementation classes, which provide their bodies. Thus, *is_a* relationship is established between the skeleton and implementation classes.

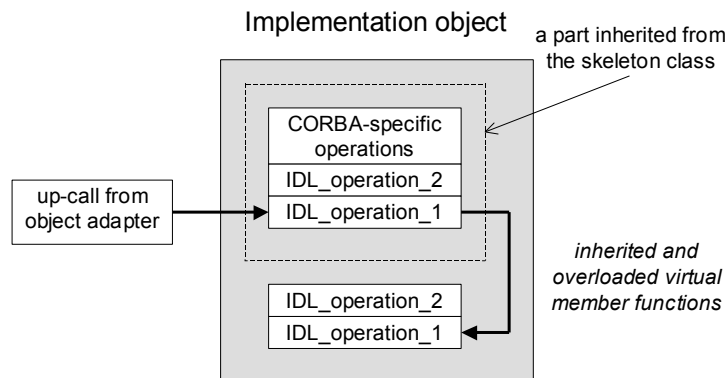


Figure 9. Inheritance-based approach

Delegation-based Approach

In the second approach, the programmer can implement the IDL operations and attributes in a class that does not inherit from the corresponding skeleton class. In this case, the IDL compiler generates a special form of a skeleton class called a TIE class. Instances of this class hold references to the associated implementation objects. Through the references method

invocations are *delegated* to the implementation objects. Thus, the *part_of* relationship is established between the TIE and implementation objects (Figure 10).

It should be emphasised, that the implementation class does not inherit any CORBA-specific classes such as skeletons and implementation objects are „tied“ to IDL interfaces at runtime. Thus, the coupling between CORBA object and implementation object in this case is much looser than in inheritance-based approach.

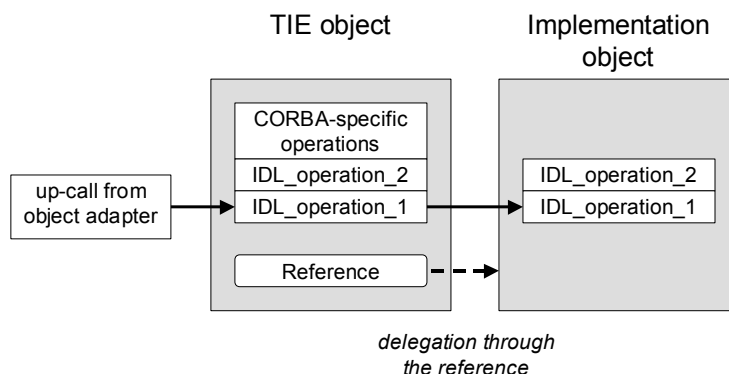


Figure 10. Delegation-based approach

2.2.6.2 Dynamic Invocation

Stub and *skeleton* are interface-specific and are statically built into client and server applications. Therefore, this method of request invocation is called *static invocation*. Another method to communication is using dynamic invocation and dispatch provided by *Dynamic Invocation Interface* (DII) and *Dynamic Skeleton Interface* (DSI) in the client and server sides, respectively. This method involves the construction and dispatch of a CORBA request at runtime rather than at compile time (as in the static approach). Because no compile time information is available, the creation and interpretation of requests at runtime requires access to services that can supply information about the interfaces and types. This information can be obtained from the *Interface Repository*, a service that provides runtime access to IDL definitions (Figure 4).

Whether a CORBA object is being incarnated by a DSI servant or by a type-specific servant, is transparent to its clients. Two CORBA objects supporting the same interface may be incarnated, one by a DSI servant and the other with a type-specific servant. Furthermore, a CORBA object may be incarnated by a DSI servant only during some period, while the rest of the time is incarnated by a static servant.

DII and DSI interfaces can also be used in combination with their static counterparts. For example, DII interface, which is used by clients to invoke operations not known at compile time, can be used together with static skeletons in the server. Just as an object implementation cannot determine whether it was invoked through a static call or through the DII, a client cannot determine whether a request was fulfilled by a static implementation or through the DSI. This is illustrated in Figure 11.

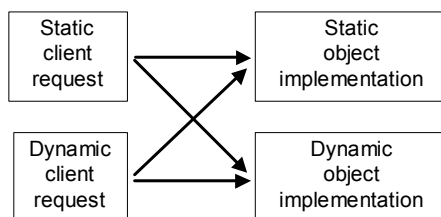


Figure 11. Mixing of static and dynamic interfaces

2.2.7 Portable Object Adapter (POA)

Until version 2.1, CORBA contained specifications only for the *Basic Object Adapter* (BOA) [OMGorb97]. The BOA was the original CORBA object adapter, and its designers felt that it would suffice for the majority of applications, with other object adapters filling only niche roles. However, CORBA did not evolve as expected because of the following problems with the BOA specification. The Portability Enhancement RFP [OMGpor95] issued by the OMG in 1995 to address these issues contained a seven-page listing of problems with the BOA specification. As result, CORBA version 2.2 introduced the *Portable Object Adapter* (POA) to replace the BOA. Addressing the full gamut of interactions between CORBA objects and programming language servants while maintaining application portability, the POA specification represent significant extension and improvement of the BOA specification.

This section provides the rationale for the POA design, defines terminology and basic concepts that will be used in subsequent sections.

2.2.7.1 POA Abstract Model

The model supported by the POA is a specialisation of the general object model described in the previous sections. Most of the elements of the CORBA object model are already presented, but there are also some new concepts related to the POA.

The abstract model supported by the POA redefines and introduces some new terms.

Servant – A *servant* is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with multiple servants.

Object Reference – An *object reference* in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an *Object ID* and a POA Identity. Note that a concrete reference in a specific ORB implementation will contain more information, such as the location of the server and POA in question.

Object ID – An *Object ID* is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. *Object ID* values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. *Object ID* values are hidden from clients, encapsulated by references. *Object IDs* have no standard form; they are managed by the POA as uninterpreted octet sequences.

POA – A *POA* is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.

Policy – A *Policy* is an object associated with a POA by an application in order to specify characteristics of that POA. Policies control a variety of options related to the management of objects and affect how an ORB processes requests on objects implemented in the POA.

POA Manager – A *POA manager* is an object that encapsulates the processing state of one or more POAs. Using operations on a *POA manager*, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the *POA manager* to deactivate the POAs.

Servant Manager – A *servant manager* is an object that the application developer can associate with a POA. The ORB will invoke operations on *servant managers* to activate servants on demand, and to deactivate servants. *Servant managers* are responsible for managing the association of an object (as characterised by its *Object ID* value) with a particular *servant*, and for determining whether an object exists or not. There are two kinds of servant managers, called *ServantActivator* and *ServantLocator*. The type used in a particular situation depends on policies in the POA.

Adapter Activator – An *adapter activator* is an object that the application developer can associate with a POA. The ORB will invoke an operation on an *adapter activator* when a request is received for a child POA that does not currently exist. The *adapter activator* can then create the required POA on demand.

2.2.7.2 POA Architecture

This section describes the architecture of the abstract model implied by the POA, and the interactions between various components.

A CORBA object adapter has three key functions.

1. Create object references, which allow clients to address objects.
2. Ensure that each target object is incarnated by a servant.
3. Take requests dispatched by a server-side ORB and further direct them to the servants.

Dispatching the incoming requests to the target CORBA objects the POA deals with *Object IDs* and *active servants*. By *active servants*, we mean implementation objects that exists in memory and has been presented to the POA with one or more associated object identities (*Object IDs*). There are several ways for this association to be made.

1. If the POA supports the `RETAIN` policy, it maintains a map, labelled *Active Object Map* (AOM), that associates *Object IDs* with *active servants*, each association constituting an active object (Figure 12).
2. If the POA has the `USE_DEFAULT_SERVANT` policy, a *default servant* may be registered with the POA.
3. Alternatively, if the POA has the `USE_SERVANT_MANAGER` policy, a user-written *servant manager* may be registered with the POA. If the AOM is not used, or a request arrives for an object not present in the AOM, the POA either uses the *default servant* to perform the request or it invokes the *servant manager* to obtain a servant to perform the request. If the `RETAIN` policy is used, the servant returned by a *servant manager* is retained in the AOM. Otherwise, the servant is used only to process the one request.

In POA specification, the term *active* is applied equally to *servants*, *Object IDs*, and *CORBA objects*. An *object* is active in a POA if the POA's Active Object Map contains an entry that associates an *Object ID* with an existing *servant*. When this specification refers to *active Object IDs* and *active servants*, it means that the *Object ID* value or *servant* in question is part of an entry in the AOM. An *Object ID* can appear in a POA's Active Object Map only once.

2.2.7.3 Request Processing

Figure 12 presents the schema of request processing with POA. A request must be capable of conveying the *Object ID* of the *target object* as well as the identification of the POA that created the target object reference. When a client issues a request, the ORB first locates a correct server (perhaps starting one if needed) and then it locates the suitable POA within that server.

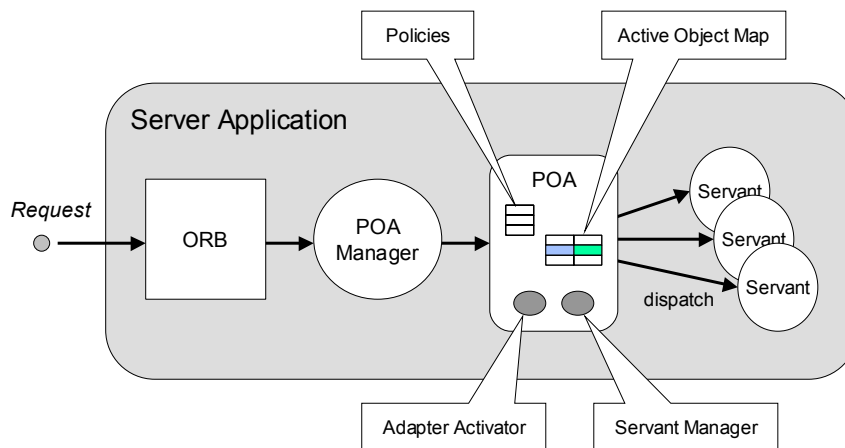


Figure 12. Request dispatching in POA

If the POA does not exist in the server process, the application has the opportunity to re-create the required POA by using an *adapter activator*. An *adapter activator* is a user-implemented object that can be associated with a POA. It is invoked by the ORB when a request is received for a non-existent child POA. The *adapter activator* has the opportunity to create the required POA. If it does not, the client receives the exception.

Once the ORB has located the appropriate POA, it delivers the request to that POA. The further processing of that request depends both upon the policies associated with that POA as well as the object's current state of activation. If the POA has the `RETAIN` policy, the POA looks in the Active Object Map to find out if there is a servant associated with the Object ID value from the request. If such a servant exists, the POA invokes the appropriate method on the servant. If the POA has the `NON_RETAIN` policy or has the `RETAIN` policy but did not find a servant in the Active Object Map, the POA takes the following actions.

- If the POA has the `USE_DEFAULT_SERVANT` policy, a default servant has been associated with the POA so the POA will invoke the appropriate method on that servant. If no servant has been associated with the POA, the POA raises the `OBJ_ADAPTER` system exception.
- If the POA has the `USE_SERVANT_MANAGER` policy, a servant manager has been associated with the POA so the POA will invoke `incarnate()` or `preinvoke()` on it to find a servant that may handle the request. The choice of method depends on the `NON_RETAIN` or `RETAINS` policy of the POA. If no servant manager has been associated with the POA, the POA raises the `OBJ_ADAPTER` system exception.
- If the `USE_OBJECT_MAP_ONLY` policy is in effect, the POA raises the `OBJECT_NOT_EXIST` system exception.

It is important to note, that POA objects are not persistent. No POA state can be assumed to be saved by the ORB. It is the responsibility of the server application to create and initialise the appropriate POA objects during server initialisation or to set an *adapter activator* to create POA objects needed later. Creating the appropriate POA objects is particularly important for persistent objects, objects whose existence can span multiple server lifetimes. To support an object reference created in a previous server process, the application must recreate the POA that created the object reference as well as all of its ancestor POAs. To ensure portability, each POA must be created with the same name as the corresponding POA in the original server process and with the same policies. (It is the user's responsibility to create a POA with these conditions.)

2.2.8 CORBA Implementations: Orbix and ORBacus

To evaluate the potential of the CORBA/ODBMS integration in practice, we had to test different integration strategies on real-life products. At present, there are numerous commercial CORBA implementations available on market. We have had experience with two of them – *Orbix v.2.3c* [ITorbix97] developed by IONA Technologies, and *ORBacus v.4.1.0* [ITorbacus01] developed by Object Oriented Concepts, Inc. (OOC). To expand IONAs leadership in the object middleware market and to add embedded/real-time functionality to the Orbix product family, in February 2001 IONA Technologies has acquired OOC Inc. Therefore, initially developed by OOC Inc., now ORBacus is distributed by IONA Tech.

The both implementations belong to the list of the most popular commercial CORBA products, which realise all features important for CORBA/Database integration. The basic difference between them is that Orbix implements a CORBA 2.0 standard specification [OMGorb97], which specifies only *Basic Object Adapter* (BOA) that functionality is not enough for the CORBA/Database integration. Therefore, to make the integration possible, Orbix introduces some proprietary extensions to the CORBA model – *loaders* and *filters*. Later, these extensions have been standardised by the OMG as parts of *Portable Object Adapter* (POA) – *servant managers* and *portable interceptors*, respectively [OMGorb01]. The POA constitute a part of an actual CORBA 2.6 specification implemented by the ORBacus. Below we will briefly describe those of them that are relevant to the CORBA/ODBMS integration subject.

2.2.8.1 Object Loading and Activation at Runtime

Object references for persistent objects should continue to be valid beyond the server's lifetime. For example, they must preserve their validity when the server makes shutdown and starts again. However, a CORBA object must exist before it can have its operations invoked. This mean that a servant implementing the persistent object must be *activated* in the object adapter before it can handle any request invocations.

For BOA adapter *object activation* is synonymous with *servant instantiation* (creation, construction). The instantiation of the servant can be accomplished in two different ways.

1. A servant is instantiated by the server application (within server mainline) or by the mean of a special *object factory*.

```
// IDL
interface ObjectFactory {
    Object create(...);
};
```

2. A servant is instantiated by a special object locating routine – *loader*, which is launched on the server side when the referenced object is not found in the server-side active objects table. The routine usually reads the object state from the persistent store and, using it, constructs the object.

In Orbix the loading operation will be performed by instantiations of a *loader* class. A programmer defines a custom loader class by inheriting a built-in `LoaderClass` and redefining a virtual member function `load()` to provide application-specific behaviour. The object adapter calls the `load()` method whenever a reference to a local object that is currently inactive enters the server's address space (in CORBA terminology, an *object fault* occurs). It is necessary to note that an object reference can enter the server's address space in the following ways.

- as the target of a request;
- as an `in` argument of a request;

- as an output argument (`out` or `inout`) or return value of a request if server acts as a client of another server;
- as a result of conversion of a string into object reference (`string_to_object`);
- as a result of proprietary `_bind` operation's invocation.

The simplified schema of object loading and activation in Orbix is illustrated in Figure 13. The `load()` method (4) uses the input object reference to identify the correct object to be loaded¹⁵. Sometimes loader may also need to detect the loaded object's initial parameters (in C++ terms, the arguments of the constructor). If the loader succeeds, it dynamically rebuilds the CORBA object at runtime (5). As an output, `load()` returns a reference to a found or newly instantiated servant. In the other case, if the loading failed, an exception will be raised. After that, a CORBA object must be *activated* (7) in the object adapter before the first request for that object is delivered. Therefore, for BOA adapter *object activation* is synonymous with *object instantiation* (creation, construction) and will be done implicitly during instantiation of the servant.

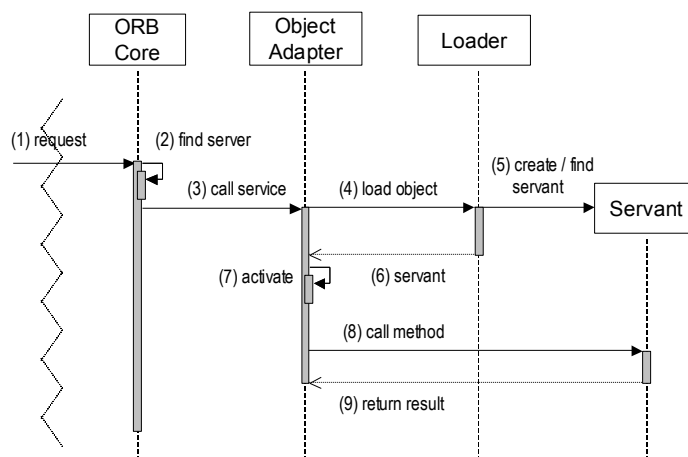


Figure 13. The schema of object loading and activation at runtime

The *loaders* proved to be indispensable in a certain kind of CORBA applications. It is the only way enabling persistence of CORBA server objects. Therefore, OMG standardised object loading routines and added this feature to the POA specification [OMGorb01]. The POA supports object loading with the help of *servant managers* that will be discussed based on the implementation in ORBacus [ITorbacus01]. In contrast to the BOA, the POA changes the scheme of object activation introducing several new choices.

1. An application can activate objects without creating any servants.
2. An application can explicitly activate an object. Implicit object activation is also possible.
3. An application can supply one of two types of *servant manager* objects that can dynamically supply servants on a per-request basis: *ServantActivator* and *ServantLocator*.
4. An application can provide a *default servant* that will be used if the target object is not currently incarnated by any other servant.

To support object loading, a POA has to have a `USE_SERVANT_MANAGER` policy and *servant managers*, which actively participate in this process. A *servant manager* is a callback object that the application registers with a POA to assist or even replace the function of the POA's own

¹⁵ Typically, loader uses Object ID as a key to find appropriate object's state stored in persistent store (e.g. database or file).

Active Object Map (Figure 12). When the POA attempts to determine the servant associated with a given target object, it calls back to the application's servant manager to get the servant. Servant managers have the same schema of object loading as Orbix loaders (Figure 13). Because a servant manager is also an object, it should be created and activated before it can be registered with the POA. There are two types of servant managers.

- For a POA with the `RETAIN` value for the `ServantRetention` policy, your servant manager object must support the `ServantActivator` interface.
- For a POA with the `NON_RETAIN` policy value, your servant manager must support the `ServantLocator` interface.

When a POA with the `RETAIN` policy value receives a request for a target object, it consults its Active Object Map to see whether a servant is already available for that object. If none is found but the application has registered a `ServantActivator` with the POA, the POA invokes the `incarnate()` operation of the `ServantActivator` object, passing it both the Object ID of the target object and a reference to itself. The implementation of the `incarnate()` operation either creates a suitable instance of a servant and returns it, raises a system exception, or raises a `ForwardRequest` exception.

For POAs with the `USE_SERVANT_MANAGER` and `NON_RETAIN` policy values, a servant manager must support the `ServantLocator` interface. The interface provides the `preinvoke()` and `postinvoke()` operations.

A POA with the `NON_RETAIN` policy value does not store object-to-servant associations in its Active Object Map, so it must invoke its `ServantLocator` for each incoming request. It first invokes `preinvoke()` to obtain a servant to dispatch the incoming request. After the request returns, the POA invokes `postinvoke()` to allow the `ServantLocator` to perform servant cleanup or other post-invocation functions. As far as the POA is concerned, the servant returned by `preinvoke()` is used only for a single request.

It should be noted, that neither Orbix nor ORBacus do provide any built-in support for *deactivating* of activated objects. To avoid server's memory overloading, programmers should apply own mechanisms (see, for example the *Evictor* pattern [Con98, HV99]).

2.2.8.2 Filtering Operation Calls

Sometimes CORBA programmers need some procedures to be triggered by incoming or outgoing messages. Such procedures – called *filters* or *interceptors* – might be performed between the consecutive regular CORBA actions. For example, filters are used in various applications to perform security checks, set debugging and measurement traps or synchronizing interaction with other applications.

Orbix provides interceptors facility by introducing *filters* – special routine launching user-defined procedures before or after individual communication events in system. There are two types of filters: *per-process* filters (applying to all objects in a particular process's address space) and *per-object* filters (applying only to individual objects). Per-object filters can be applied only to server-side objects, whereas per-process filters can be used on both client and server sides. Below per-process filters will be described in more details.

Each *per-process* filter class monitors eight points of the client-server communication chain: four points at the client side (before and after outgoing request is marshalled and before and after incoming reply is unmarshalled) and four points at the server side (before and after incoming request is unmarshalled and before and after outgoing reply is marshalled).

OMG has standardised this concept in CORBA. It appeared the first time in version 2.2 as the concept of *portable interceptors* [ITorbacus01, OMGorb01].

2.3 Motivations to CORBA/ODBMS Integration

The previous chapters described CORBA and Object-Oriented Databases as being separate technologies. Indeed, CORBA and ODBMSs are existing on the market and successfully used in different domains as autonomous tools. Their *integration* can be defined as a process applied to bring both technologies work together as a single CORBA/ODBMS system.

Each party of the integration is free to interpret it in own way. Thus, the „integration“ can be considered from the CORBA perspective. In this case, CORBA server objects are stored and managed by the ODBMS system. Alternatively, the „integration“ can be considered from the ODBMS perspective. In this case, ODBMS objects are made remotely accessible by CORBA clients.

In practice, different approaches adopted on different levels of abstraction can be applied to attain the integration [VA94]. All those approaches imply the existence of an additional *mediating tier* operating between both parties of the integration. The resulting CORBA/ODBMS system is composed from three tiers, where CORBA operates as a front-end and the database operates as a back-end (Figure 14). Effectiveness, flexibility and, as a result, potential benefits determining the motivations for the integration depend on the properties of the mediating tier. Still, what are the motivations for CORBA/ODBMS integration? Let us analyse them separately taking the viewpoints of the both parties of integration.

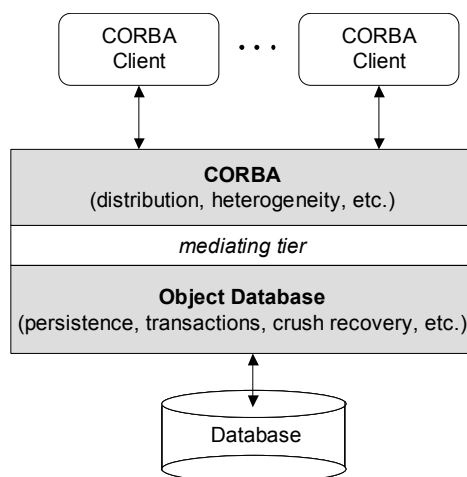


Figure 14. Three-tier architecture of the CORBA/ODBMS system

2.3.1 Motivations of CORBA

Summarising a discussion about CORBA, now we put a detailed look at the advantages and limitations of CORBA.

1. CORBA supports both distribution and object orientation.
2. CORBA provides a high degree of interoperability. This insures that distributed objects built on top of different CORBA products can communicate with each other. Large companies do not need to mandate a single CORBA product for all their developments.
3. CORBA supports many existing languages. CORBA also supports mixing these languages within a single distributed application.
4. CORBA is an industry standard. Over 600 companies back CORBA, including hardware companies, software companies, and cable companies, phone companies,

banks, etc. This creates competition among vendors and ensures that quality implementations exist. The use of the CORBA standard also provides the developer with a certain degree of portability between implementations. Note that the application source is not 100% portable between different CORBA products.

It is important to note that the entire request invocation is *transparent* to the client, to which a request to a remote object looks like an ordinary method invocation on a local C++ object. Although the ORB can tell from the object reference that the target object is remote, the client cannot. There is nothing in the object reference token that the client holds and uses at invocation time that identifies the location of the target object. This ensures *location transparency* – the CORBA principle that simplifies the design of distributed object computing applications. In particular, the request invocation has the following characteristics.

Location transparency. The client does not know or care whether the target object is local to its own address space. It is implemented in a different process on the same machine, or in a process on a different machine.

Server transparency. The client does not need to know which server implements which objects.

Language independence. The client does not care what language is used by the server. CORBA supports many existing languages and allows mixing them within a single distributed application.

Implementation independence. The client does not know how the implementation works. The client sees the same consistent object-oriented semantics regardless of how objects are implemented in the server.

Architecture independence. The client is unaware of the CPU architecture that is used by the server and is shielded from such details as byte ordering and structure padding.

Operating system independence. The client does not care what operating system is used by the server. The server may even be implemented without the support of an operating system – for example, as a real-mode embedded program.

Protocol independence. The client does not know what communication protocol is used to send messages. If several protocols are available to communicate with the server, the ORB transparently selects a protocol at runtime.

Transport independence. The client is ignorant of the transport and data link layer used to send messages. ORBs can transparently use various networking technologies such as Ethernet, ATM, token ring, or serial lines.

Through the integration with database management systems, modern CORBA tools receive new features, which are necessary for mission-critical applications. In particular, limitations of CORBA systems can be removed or at least considerably reduced.

Server memory management. One of the inherent problems of CORBA (as well as other distributed object-oriented platforms) is the server objects' garbage collection. Server objects (previously made to carry out remote clients' requests) not used by any client should be disposed in order to prevent the server's memory leak. The reference counting provided by `duplicate()` and `release()` operations of `Object` interface concerns only local memory management. It allows CORBA to identify whether a *local* proxy object on the client side or whether the implementation on the server side is no longer referenced within its *local* address space. This should be not mixed-up with distributed reference count, which would correlate the lifetimes of client- and server-side objects. For example, calling `release` on the client-side proxy object does not affect anyhow the implementation object on the server-side. As a result, the

problem of server memory management must be resolved by some extra mechanisms¹⁶ (see, for example, [Con98]).

Persistence. In many applications the lifetime of objects must exceed the lifetime of the process, which created that objects. Although attempts have been made to define the *Persistent Object Service* (POS) [OMGpos94, OMGpos00], they have finally failed [KPT96a, KPT96b] and now a new standard *Persistent Store Service* (PSS) [OMGpss99] is adopted to be enclosed in CORBA 3. Unfortunately, the current release of the standard does not contain any well-defined ways to provide persistence to CORBA objects.

Crash recovery. Many mission-critical applications need mechanisms for crash recovery guaranteeing data integrity. CORBA does not have such mechanisms and most of commercial CORBA implementations do not provide them.

Multithreading and concurrency control. Many of the modern CORBA implementations support multithreading. When server processes requests coming from multiple clients simultaneously, the issue of concurrent access to server objects arises. The problem is become worse as the number of clients and server objects increases threatening the CORBA system scalability. In addition, the CORBA specification completely ignores the issues of multithreading leaving them to CORBA implementers [SV9798]. The situation was somewhat improved with the commercial release of CORBA implementations supporting POA (see Section 2.2.6), which pays more attention to issues of multithreading.

2.3.2 Motivations of ODBMSs

Summarising all new object-oriented features to traditional database technology, here are the most essential fields where ODBMSs add cardinal improvement with respect of the relational model.

- ODBMSs solve the *impedance mismatch* problem [CM84] by the integration of the programming and database environments. A database application can be written in a single and computationally complete language. There is no need for an additional data description language – capabilities of a database programming language are used to outline data structure. In addition, a data manipulation language could be substituted by a minimal set of new operations usually added to programming language.
- ODBMSs support user-defined data types, i.e. data structures together with operations on them. Classes are used to represent new data types. All database management systems, except the ODBMSs, are able to handle simple types only and in some cases simple objects (e.g. BLOBs) or collections. The semantic richness of the object-oriented approach enables data models to be built, which more closely reflects real-world environment they model.
- In contrast to *value-based* RDBMSs, ODBMSs are *identity-based* systems. In other words, they support the concept of *object identity*: an object has an identity independent of its value. Object identity in turn is used to support direct object relationships, i.e. an object can have an attribute whose value is another object (or pointer to another object), and thus allowing nested data to be naturally represented in the database.

¹⁶ Although various techniques of garbage collection for server-side objects currently exist, none of them is optimal enough to be incorporated into CORBA.

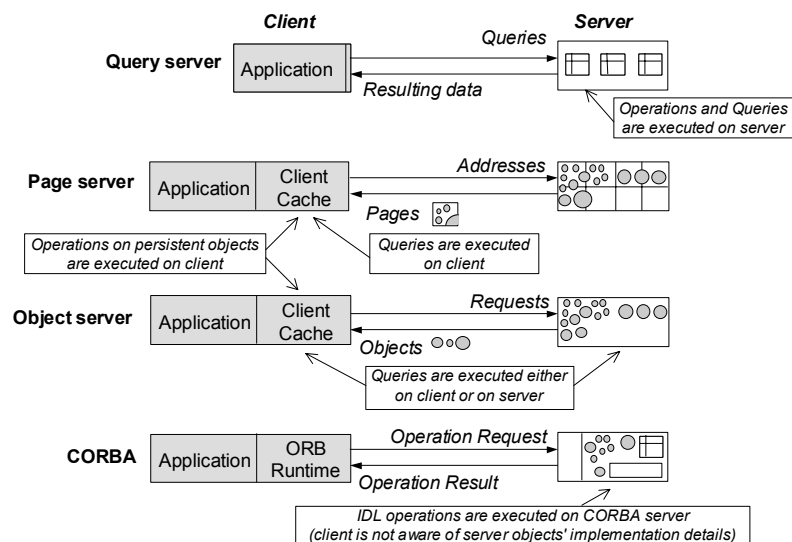


Figure 15. Comparison of client/server architectures of RDBMS, ODBMSs and CORBA

Despite providing these features, modern ODBMS products have some substantial deficiencies, which may deter many potential users from giving up the obsolete relational systems in object-oriented favour. Thus, the profits of ODBMSs from the integration with the CORBA environment can be even more substantial and various than for CORBA.

Like relational DBMSs, almost all ODBMSs are designed as client/server systems. Figure 15 shows four major architectures have been established that differ significantly in the extent the workload and „intelligence“ is distributed between client and server [Heu92].

Query server: This architecture concentrates most of the workload at the server-side. The server processes queries and transmits the resulting data (i.e. a collection of objects) to the client. This architecture performs well, if a typical query asks for a small subset of data. In this case, the network load is small. As most of the work is done by the server, the requirements for the client hardware are small (thin client).

Page server: In this architecture, the sole purpose of the server is to give access to pages (i.e. the smallest storage unit) that contain objects needed by the client, to store pages with new or updated data and to assist in locking the pages. The client has to build objects from these pages, has to process queries etc. Thus, the most of the work in this architecture is done by the client. This approach works well and avoids server-side bottlenecks unless selective queries have to be done. In this case, the transmission of the numerous pages to the client causes a lot of network traffic, even though most of these pages might not match the query conditions at all.

Object server: In this architecture, the client requests entire objects. Transactions and locking mechanisms use objects (instead of pages) as units. This requires much more „intelligence“ at the server-side. Depending on the implementation, querying can be done either at the client or at the server.

CORBA: In the contrast to the previous models, the distribution model of CORBA is much more flexible and can be used for various client/server architectures. Thus, CORBA supports *query (or operation) shipping*, which is typical for the first architecture, and *data (or object) shipping*, which is typical for the other two architectures. The weights of clients and servers are not fixed and depend on the concrete applications. Moreover, in the contrast to all previous architectures, CORBA allows a more loose coupling between clients and servers. In the previous architectures the clients and

servers are tightly coupled that is forcing database clients to have precise knowledge of how persistent objects are logically stored in the database (database schema). In CORBA, the clients are not tied to concrete servers, but are free and can communicate with arbitrary servers. All that CORBA clients need to know for communication with the server are the IDL interfaces of the objects accessed.

Almost all RDBMSs are based on query servers. For ODBMSs, this type of server is rare because of two reasons: firstly, queries do not have the same priority as for RDBMSs. (Many ODBMS applications simply traverse items of an object hierarchy and do not need queries at all.) Secondly, query servers have turned out to be the bottleneck of systems with many concurrent users. This leaves page and object servers for ODBMSs: [CFZ94] explains why page servers outperform object servers in the most test scenarios. On the other hand, [HPH97] describes an example (a commercial application that previously used an RDBMS) where an object server outperformed a page server significantly. In many ODBMS implementations, the distinction between these two architectures becomes less clear anyway. The ideas from both architectures are frequently mixed (e.g. by combining a page server with object locking or by giving page-wise access to large objects in an object server). Thus, the integration with the CORBA environment, or, in other words, using CORBA vendor-neutral object transportation and method invocation abstractions to access database objects, can solve or at least substantially diminish the limitations of ODBMS.

Single-language environment of ODBMSs can be considerably enhanced. CORBA-compatible database clients may be implemented using any programming language for which an IDL mapping is defined and realised. Moreover, clients are not limited to the operating systems supported by particular ODBMS vendors and can run on a wide range of CORBA-supported software platforms.

Lack of support for views in ODBMSs can be alleviated through „IDL views“. By integrating with CORBA, ODBMSs can be provided a kind of „object views“ mechanism. Indeed, since CORBA clients communicate with server objects only via defined IDL interfaces, the „image“ seen by the clients can be effectively handled through the IDL interfaces manipulation. First, only a subset of objects stored in the database can be visible to the clients. Moreover, only some members and methods of a persistent object can be exposed to clients via the related IDL interface.

Tight coupling with client applications in ODBMSs can be replaced by the loosely coupling with CORBA clients. As opposed to ODBMS clients the CORBA clients can operate without any knowledge about the logical layout of objects stored in the database (database schema), so most frequently the evolution of the database structure will have no impact on client applications.

Limited distribution of ODBMSs can be substantially broadened. In contrast to traditionally heavyweight ODBMSs client processes, where most, if not all, data processing takes place, the CORBA-based clients can be essentially lightweight. Thus they demand less computing power and slower communication links. In fact, in one system, clients of both types can coexist, which gives an application developer a great potential to adapt to specific needs/conditions of a particular application.

Limited security of ODBMSs can be improved. Making clients access the server objects through a CORBA system is a perfect solution for applications demanding higher level of data security as well as unified behaviour of server objects. Indeed, CORBA clients use remote method invocation to access server objects and, therefore, cannot change their behaviour. As a result, integration with CORBA allows C⁺⁺ ODBMSs to be real „object databases“, rather than „object state databases“, as they presently are.

Poor authorisation mechanisms of ODBMSs can be refined. Thanks to the integration with CORBA a manipulation of modes of clients' access to objects stored in the database can be performed. The application developer can freely choose whether the particular client should be aware of the target object's persistence, whether to show up the specific database operations such as open/close database, begin/commit/abort transaction, query database and other. Such a great flexibility allows to support significantly better authorisation mechanisms for the clients.

In addition, distributed transactions management can be greatly facilitated using CORBA's Object Transaction Service (OTS) [OMGtran01]. Based on the widely accepted non-object Distributed Transaction Processing (DTP) X/Open reference model, the OTS replaces *XA* and *TX* interfaces by a set of CORBA interfaces declared in IDL, mapping TP robust monitor technology to the CORBA-managed object world. Thus, OTS introduces reliable ACID transactions in distributed environment connecting applications operating on different software platforms and implemented by different programming languages. By integrating with CORBA, heterogeneous ODBMSs are able to interact with each other in the distributed environment like any other CORBA-compliant applications.

Finally, the price of the most refined CORBA products usually makes no more than 5-15% of the cost of ODBMS. The extra costs of the integration are insignificant comparing with its huge benefits. Thus, the integration can stimulate the further popularisation of CORBA use in ODBMS-based systems.

2.3.3 The Benefits of CORBA/ODBMS Integration

As described in the previous sections, the potential benefits, the integration brings to CORBA and ODBMS systems, depend heavily on the particular technique applied to achieve the integration. Summarising, now the most important benefits of the integration are outlined.

In the integrated CORBA/ODBMS system, CORBA and ODBMSs play different roles. In the OMG environment, an ODBMS can support the definition, creation, and manipulation with the services of persistence, transactions, recovery, and concurrent sharing for application objects ranging from the smallest units to the largest. Many applications require these services to provide transparent distribution in a heterogeneous mixture of platforms and other services such as versioning and security, all from a single product. ODBMS is best thought of in terms of the services that it provides, and not according to any particular implementation. Radically different implementations are possible, each offering different levels of services and trade-offs.

In contrast, the ORB provides a larger-scale set of services across heterogeneous vendors and products. For example, it allows clients to use multiple data storage products. It provides behaviour invocation or method dispatch. In contrast, a given ODBMS provides a single-vendor capability and only a specific set of services rather than an arbitrary-defined set. However, those services include more complete capabilities of high-performance, fine-grained persistence that are used directly within applications to support millions of primitive objects. The ORB, when it needs persistence services, could choose to implement them with an ODBMS object storage, whose services may be invoked via the ORB.

Thus, it seems that a standard middleware platform such as CORBA can provide effective and straightforward means to relieve database drawbacks at low cost. The database community seems to get the message: in the recent months the use of middleware in database applications has become more and more popular, if not widespread. Not surprisingly, the number of reports/evaluations/documents about the issues of application of

these mechanisms to improve database functionality constantly increases [OOPSLA97, OOPSLA01].

Conversely, despite the tangible profits CORBA applications potentially gain from the integration, the CORBA community seems to rush towards the integration with less enthusiasm. In fact, the benefits brought by a full-fledged DBMS can be often replaced by lightweight and less expensive applications implementing related CORBA services (Persistence, Transaction, Query, Concurrency Control etc.). The important conclusion can be drawn that the CORBA/ODBMS integration is much more meaningful and profitable for databases. That fact cannot be ignored when the proper techniques/methods are being chosen to achieve maximal effectiveness and flexibility of the resulting CORBA/ODBMS system.

Chapter 3

The Evolution of CORBA/ODBMS Integration Techniques

This chapter defines the fundamental issues of the CORBA/ODBMS integration and describes the ways each strategy tackles with particular integration issues, as well as presents their detailed evaluation.

The two first sections specify the basic strategies and the general scheme of the CORBA/ODBMS integration. The subsequent sections cover the evolution of the design ideas and tools: early database adapters and wrappers are described in brief, whereas special attention is paid to approaches based on the idea of Object Database Adapter. The remainder of this chapter contains a comprehensive discussion over the applicability of particular strategies. Finally, it conducts their detailed comparison.

3.1 Basic Integration Strategies

In this section, we present three basic integration strategies for the integration of existing database management systems with CORBA. They all use CORBA for database integration, but each follows a distinct flavour of programming style. Though, other combinations make also sense in certain situations, we have chosen those three discussed in the following to demonstrate the consequences of basic design decisions for the integration.

3.1.1 Server-side Query Approach

This integration strategy is oriented on all conventional database management systems providing a server-side query functionality similar to RDBMSs. In this approach CORBA clients communicate with the database via a set of queries. Clients submit their queries as strings or use hardcoded, fixed queries, and receive results as strings or collections of strings. The middleware has no notion of the schema of the underlying database. The approach was used in many research projects. For example, it was studied by integrating a deductive object-oriented knowledge database *ConceptBase* with CORBA [BS95].

The MIND project [DD96, DD98] uses CORBA as the middleware for a system of federated databases, both relational and object-oriented. The authors describe several design decisions made for this purpose: the use of an unshared server activation mode for parallel query execution; static method invocation, because they assume a static database schema; and the decision to only register the database gateway as a CORBA object, to avoid extensive IDL coding and to be immune against schema evolution. Hence, they took an approach similar to our „server-side query“ scenario. They do not treat CORBA services, probably because they were not yet readily specified during the writing of these papers.

Although this approach is old-fashioned, it has a number of advantages. Using queries clients can use the full power of the query language of the database. As there is no schema representation encoded in the middleware layer, it is immune against schema changes in the database. Furthermore, queries are transmitted „as it“, and can be optimised by the database.

Disadvantages include the tiresome parsing of all results on the client side. This becomes a challenging task, if the transmitted data types are for instance multimedia objects such as images or audio data, or if an integration of multiple databases is needed. Therefore, this approach works best when queries must deal with large numbers of objects in processing the query, but the results can be expressed in relatively simple data types. This is true about the existence queries, which may scan large amounts of data and return boolean results, and the counting queries, which return simple data structures, or financial queries returning time-series data which may be amenable to IDL data structures such as sequences.

Because the ORB does not know the database objects, they cannot make use of any object service. The absence of schema data in the middleware requires that clients be developed in close collaboration with the server, passing meta information through some other proprietary „channel“, probably by human communication. This does not suit well to the general spirit of distributed object computing, where objects should be rather self describing and should be usable from any client within the network without human interference. However, this problem can be alleviated by implementing some methods that provide meta-data.

A server-side query approach is supported by CORBA's OQS in a trivial way, as it certainly will implement a query method as the Query Evaluators. The OTS also supports this approach as the OTS offers XA interoperability: the XA interface of the database is used.

3.1.2 Object Collections

Object collections describe a useful strategy for applications that do not need restructuring queries. For instance, a digital library service is usually based on queries that specify conditions on attributes of publications, such as the appearance of certain keywords in the abstract or a certain authorship. The result is a list of publications and it makes little sense to construct queries that result in a combination of the first name of an author with the ISBN number of another publication. Some experiences in the integration of a scientific database can for instance be found in [PBJ+00, RH97]. Semantic issues are treated in [BK97].

As a result, in such an environment it is possible to build a comfortable, fully typed database interface. Clients can retrieve the struct definitions through the ORB's interface repository, and, using DII, they can flexibly react on schema changes. Schema evolution only requires a re-compilation of the server. Once an object is passed to the client, reading attribute values does not require any remote calls any more.

However, this strategy does not offer full query power. It is also not clear how query conditions can be specified; using SQL or OQL is not possible. Simple, template-based mechanisms will fail to construct complex conditions that require joins between database classes. Furthermore, it is difficult to retain integrity in the database, if two clients have requested the same object and perform changes on its data at the same time. This can be done, for instance, by putting a read lock for every object that is read by a client, until this client releases the object again. The OQS cannot easily be adapted to this scenario, because the „objects“ involved are actually no CORBA objects, but simple structs.

3.1.3 Pure CORBA

With „pure CORBA“, we refer to a strategy that fully employs the object-oriented paradigm within CORBA: All database objects are represented as CORBA objects. As Section 3.1.1 shows, this makes the use of ad-hoc queries almost impossible, so we have to fall back on the class of restricted queries.

This is similar to the strategy discussed in the previous subsection. The only, but significant difference is the existence of CORBA objects instead of structs. Therefore, each access

requires a remote procedure call, which creates considerable network traffic. Consequently, CORBA services can be applied over these objects. For example, the OTS may be used to guarantee integrity in cases as described above.

The strategy allows different models of data access: the created CORBA object can either read all data from the database at once and then answer client requests from its internal data, or delegate each single access to the database server. In the former case, the integrity problem exists again (but only among objects on the same machine), while in the later case, the database automatically enforces integrity, but for the price of an increased number of accesses. In this case, query optimisation becomes virtually impossible.

Since only this integration strategy fully employs the object-oriented paradigm, which would be an advantage for integration with ODBMS systems, we will skip other strategies from the following discussion of possible design approaches.

3.2 Wrapping persistent Objects

The basic technique to attain CORBA/ODBMS integration is hand-coding of wrapper objects residing between two systems. This technique will be further referred to as *wrapper* approach [MZ95].

A database wrapper is a CORBA application server to ORB clients, and a client to the integrated database. The wrapper offers IDL interfaces to clients for reading or modifying data stored in the database, and so provides a higher abstraction layer between clients and databases. Figure 16 shows the architecture of a simple three-tier system with a database wrapper.

The database wrapper translates client calls into native calls to the DBMS, for example, using standard techniques such as JDBC or ODBC, or proprietary protocols like the *Oracle Call Interface* (OCI). The wrapper therefore must take care of the mapping between both data models making any necessary pre- and post-processing for data interchange, including client access authorisation (actions 1-4 on Figure 16). CORBA's extensions (such as loaders and filters in Orbix) were typically used by implementers for these purposes.

The basic advantage of this approach is that it is not oriented on any specific integration strategy and very flexible and therefore, it is often used for the integration of legacy systems with the CORBA environment [BS93, BS95].

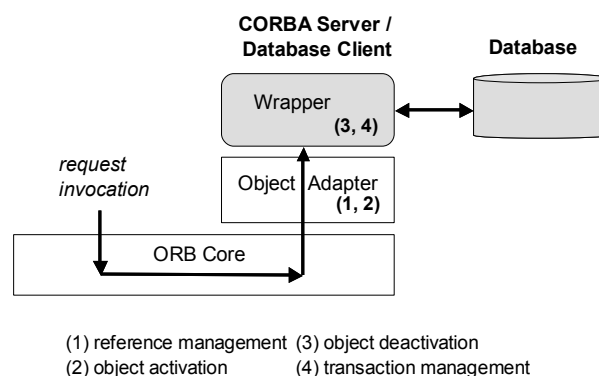


Figure 16. The functional schema of the wrapping technique

3.3 Object Database Adapters (ODA)

This idea is first time presented in Appendix B of the ODMG standard [CB97], which identifies some issues involved in using an Object Database Management System (ODBMS) with a CORBA environment, and proposes an *Object Database Adapter* (ODA) to realise the integration of CORBA with ODBMSs.

3.3.1 Early ODAs

Initially the *Object Database Adapter* (ODA) was specified as a complementary part to the *Basic Object Adapter* (BOA). It would compensate BOA’s inability to deal with persistent objects (Figure 17). In order to avoid terminology clashes, the first implementations of database adapters will be further referred to as *early adapters* or *early ODAs*.

An ODA is an object adapter for objects stored in a database, which provides a tight integration between the ORB and persistent object implementations. A normal object adapter interacts between the ORB, the generated skeletons and the object implementation. With the assistance of skeletons (or of the Dynamic Skeleton Interface), it adapts the object implementation interface to the ORB-private interface. In this description we can recognise an application of the well-known *Object Adapter* pattern [GH]+94]. Unlike the normal object adapter, ODA is aware of the persistence of its target object implementations. Unlike the BOA, ODA cannot assume that every implementation object is in memory and the adapter is prepared to deal with locks and transactions, since persistent objects are generally shared.

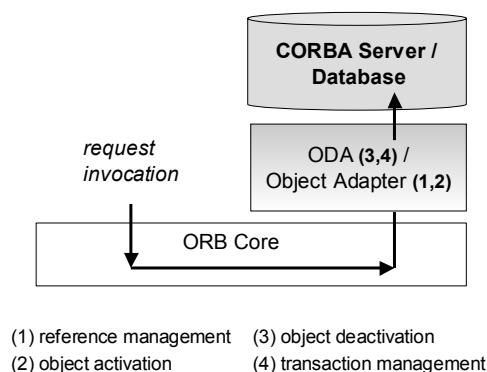


Figure 17. The functional schema of the integration using the Object Database Adapter

The main advantage of the ODA is that allowing the same object to be CORBA and ODBMS object, at the same time it provides a seamless integration of both systems. CORBA servants are stored in the database and transparently moved to CORBA server’s memory when needed. Early ODAs deal with reference management to detect the references to persistent objects (1), cooperate with the DBMS to perform persistent objects activation (2), as well as perform transaction handling (4). The task of server’s memory management (object deactivation) is executed by the underlying DBMS (3).

Thanks to ODAs the creation of a new CORBA-compatible database requires little programmatic efforts. All the above-mentioned issues are performed by the ODA transparently for application programmer. However, at the same time, the approach forces programmer to map each persistent object to an IDL interface, which may follow to unacceptable overhead in the case of many fine-grained objects (see Chapter 6.5). Besides, some CORBA-specific data are need to be stored in the database (e.g. associated TIE objects). In addition, the mechanisms of transaction management are too coarse or even absent, which result in further worsening of the overall performance.

The fact that the same object plays two roles at the same time (a database object and a CORBA servant) made this approach inappropriate for the integration of existing databases with CORBA [BBCS97, BBCS99]. If we have to deal with large data stores, which are permanently in use by multiple client applications, then it imposes additional requirements on the interoperation architecture. First, making these data stores CORBA-compliant should not disturb already existing applications. Since a schema evolution may be a very exhausting process for large data volumes, in the ideal case an arbitrary data store should be made CORBA-compliant without changing where and how the data is stored [SSC+97, BCG+98].

Providing the basic idea of possible integration ODA has found a corresponding resonance in the database research society. An ODA based on the ideas was developed as part of the Sunrise Project at the Los Alamos National Laboratory (LANL) [Rev96, RM97]. This adapter has been used by the *TeleMed* system [Fors97] since mid 1995, and is currently employed by other LANL projects as well. For example, it is used in the current *OpenEMed* system [OM03]. The adapter was implemented for two ORBs, Orbix and VisiBroker for C++, with ObjectStore as the underlying ODBMS in both cases.

The commercial examples of early ODAs were implemented, for example, for ObjectStore and Versant ODBMS [IToosa97, ITova97]. Meanwhile, the recognition that the ODA approach is not exclusive to object-oriented databases seems to have grown in industry. Object-relational mappers – systems that map C++ classes/objects into relational tables/tuples – have been employed to make relational databases appear as object-oriented ones. Because such mappers implement an ODBMS interface on top of a relational system, they extend to relational databases the applicability of the ODA approach.

An example of such a system is *ObjectDRIVER* [LDAJ00] – an ODMG-Compliant Open Object Wrapper dedicated to reusing relational databases. With *ObjectDRIVER*, a relational database can be accessible as an object database after the object schema and its mapping on top of the relational database have been defined. The object data can be queried with OQL, the ODMG query language.

ObjectDRIVER is based on a powerful database adapter, an *Abstract Data Component* (ADC) layer that makes possible to reorganise and to have access to relational data as complex data through complex views. Of course, it also allows to make complex data persistent in a relational database. Through the views all classical database operations are possible on complex data, for example, inserting, updating, deleting, retrieving and querying with OQL. A convenient interface of the ADC layer allows connecting to many object models and metamodels.

3.3.2 Orbix Database Adapter Framework

The *Orbix Database Adapter Framework* (ODAF) [ITodaf97] represents further development of ODA principles specifying a flexible architecture that could be implemented for a wide set of ODMG-compatible database management systems.

The ODAF consists of a special version of the IDL compiler and a library of abstract classes providing a rough form of the ODA's architecture. Since the library has not any real implementation, it also is not specific for any particular DBMS and allows the realisation of custom ODAs for a wide set of ODMG-compatible DBMS. Those ODAs will further be referred to as *ODAF-made adapters* (or ODAF-made ODAs)¹⁷.

¹⁷ As the ODAF is a product of IONA, and because Orbix's proprietary extensions are employed in the framework, IONA's terminology will be used to describe further ODAF features.

Using the same symbols as in the previous figures, the functional architecture of ODAF-made adapter is presented in Figure 18. According to the model applied in ODAF the CORBA implementation objects are stored in a database, while clients access their methods via transiently allocated CORBA TIE¹⁸ objects (delegation pattern [GHJ+94]), which in turn delegate method calls to appropriate persistent objects.

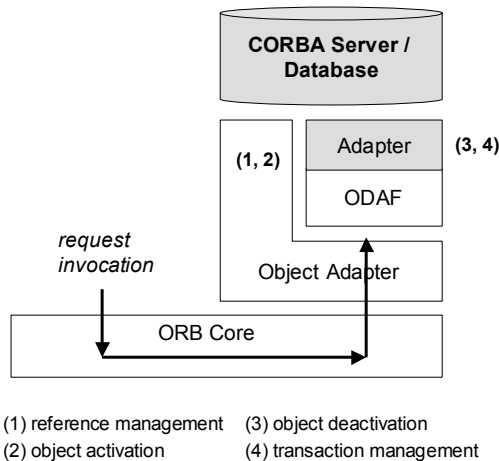


Figure 18. The functional schema of the integration using the ODAF-made adapter

For the implementation of TIE objects ODAF offers two different approaches: The Non-Persistent TIE (NPT) and Programmer-Defined TIE (PDT). The Non-Persistent TIE classes are automatically generated by the Orbix’s IDL compiler¹⁹ from interface declarations, whereas Programmer-Defined TIE objects are supposed to be designed by programmers. Therefore, they are much „smarter“ as usual TIEs. In the core, however, the basic scheme of two approaches is semantically the same and the operation of the adapter’s components is independent of the actual policy of the TIE objects’ creation.

Comparing Figure 18 with Figure 16 it should be noticed, that the ODAF-made adapter shifts the special routines used in the wrapper approach from the server application to the top of the Object Adapter. From the perspective of CORBA, ODA does not replace BOA. Instead, it adds some complementary functionality supporting the persistence of CORBA objects. Obviously, such a structure incorporates features of both early adapters and wrappers. As in the case of early adapters, the whole functionality is encapsulated in one logical piece of code compliant with the OMG standard. As in the case of the wrapper technique, an application developer has some liberty in defining the functionality of the individual ODAF-made adapter, making it more suited for the application’s specific requirements. Particularly, the implementation is separated from the interface, making the whole structure much more manageable and applicable to integration with legacy systems.

3.3.2.1 The Architecture of ODAF

ODAF is not a ready to work implementation, but a framework – a set of classes defining the interfaces of basic components and the internal structure of the adapter. Therefore, the main contribution of ODAF is a flexible architecture that could be implemented for a wide set of ODMG-compatible database management systems. The architecture defines the following components (classes).

¹⁸ It should be noted, that TIE objects are regular CORBA objects, used by the BOA for delegation.

¹⁹ More precisely, the appropriate object factories, which support TIE classes declaration and definition, are generated.

ODA control. This singleton implements the adapter's public interface accessible by ODA users. References to one *Marker Mapper*, one *TIE Manager* and one *Transaction Manager* are held by ODA control and are accessible via its (static) methods.

Marker Mapper. The unambiguous mapping between TIE (CORBA) object's identifiers (Orbix's markers) and persistent object identifiers must be performed by this component of ODA. Markers contain externalised identifiers of persistent objects in order to ensure target object location in the database. As a rule, stringified database references uniquely identifying persistent objects are used as markers of related TIE objects. Other marker mapping strategies (like using row ids, file names etc. to identify the place where the persistent object is stored) are also feasible.

TIE manager. It handles the lifetime of TIE objects. All created TIEs are registered with the TIE manager. In accordance with a certain policy (FIFO, least-used etc.) it deletes registered TIEs when needed, for example, when the number of TIE objects exceeds some limit value.

Transaction manager. It provides an interface between ODA and transaction-related operations of the database.

TIE Factory Finder. It manages TIE factories (static objects declared and defined by special macros) created by the application. Each TIE factory is specific to one particular interface and to one particular persistent class.

Loader. Being an integral part of Orbix (see Section 2.2.8), it is launched automatically by the server when the target object has to be retrieved from persistent store.

Per-process Filter. Closely cooperating with ODA control and transaction manager, it is used to manage transaction boundaries.

The ODA designer is free to introduce new marker mapping strategies, TIE management policies as well as the behaviour of the transaction manager by subclassing the related ODAF-defined classes. Installation of additional filters was also possible. Other components could be arbitrary modified so that their interfaces remain accordant to the basic ODAF internal structure.

Objects Loading and Activation

The most notable operation in ODA is an operation for the creation of a new CORBA object. The stringified database references were typically used as Object IDs (in Orbix's markers) which unambiguously identify proper target objects stored in the database. This Object ID is a part of CORBA's object reference, so the activation routine given the reference could properly set the link between the TIE object and the target persistent object. The object activation was usually performed using object activation tools like Orbix's *loader* [ITloader97]. Patterns like IONA's *evictor* [Con98] and/or *garbage collector* [ITclean97] were practicable in implementing the object deactivation mechanism. Figure 19 presents a simplified schema of steps taken by the ODA in order to handle requests targeted to non-active objects.

In accordance with Orbix's object activation mechanism (see Section 2.2.8), the *loader* receives from Orbix's runtime the marker of the target object and contacts the *Marker Mapper* to convert the marker in a database reference. Having the marker, the *loader* can contact the *TIE Factory Finder* and find a proper *TIE factory*, which is used to construct corresponding *TIE objects*. As input parameters, the factory receives the database reference, which is used to uniquely identify the TIE object. The newly created TIE has to register itself at the *TIE Manager*. Once a TIE (CORBA) object is created, it is implicitly *activated* by BOA and can direct operation calls to the tied implementation object. In critical situations, for example, if

the pool of active TIEs is full, TIE objects can be deleted by the TIE manager (dashed arrow).

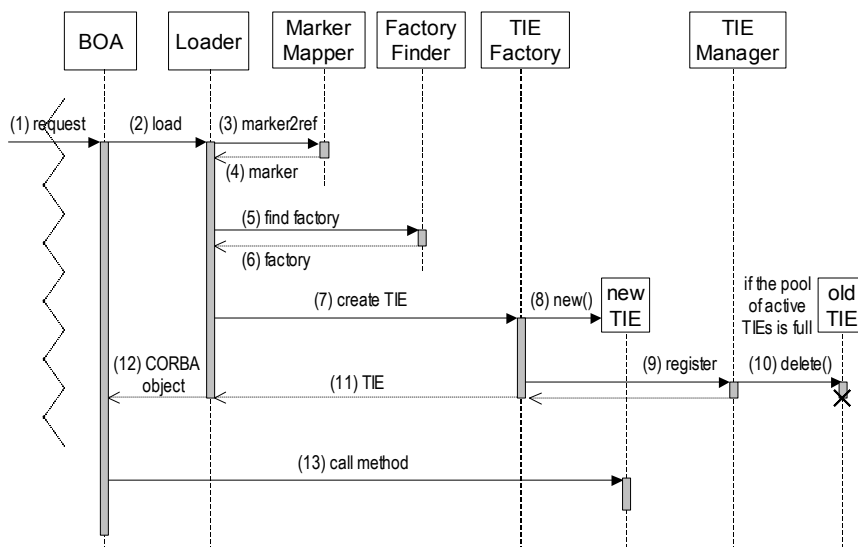


Figure 19. TIE object creation process

Transaction Management

Responsibilities for transaction management in ODAF are carried by the *ODA control* and the *Transaction manager*. The *ODA control* decides when a transaction must be started and when this transaction must be finished. The *Transaction manager* begins and ends transactions at the control's request.

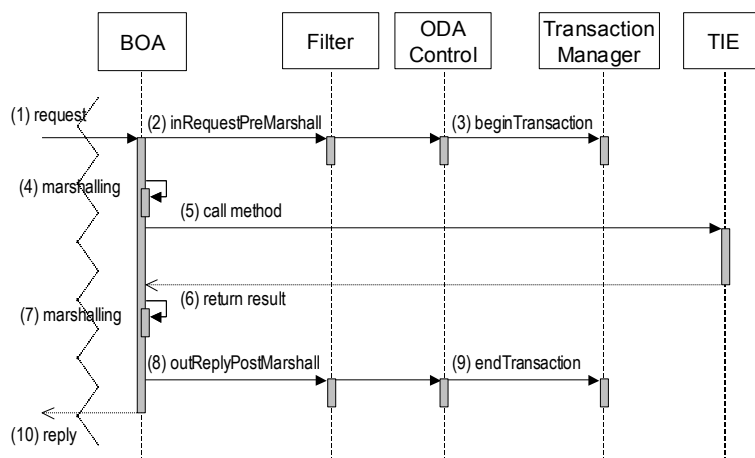


Figure 20. Per-operation transaction style

ODAF provides two styles of transactions: *per operation* (*perOp*) and *phased*. In the default *perOp* style the control starts a transaction when a request arrives and ends this transaction when the request ends (see Figure 20). With the help of a filter, the transaction manager starts transactions before marshalling of the arguments and ends this transaction after marshalling the results.

Alternatively, in the *phased* style the control starts a transaction when a request arrives and when there is no active transaction, and ends this transaction only when the programmer requests it (see Figure 21). The programmer can switch from the *perOp* style to the *phased* style and vice-versa, by calling `transactionStyle()` on the ODA control. If a transaction is started, then consecutive operation calls will be done in boundaries of the existing transaction

without starting/ending transactions for every call (dots in Figure 21). For all other clients, the access to the used persistent objects will be blocked until the end of the running transaction. To end a phased-style transaction, the programmer must call `endPhase()` on the ODA control. However, `transactionStyle()` and `endPhase()` operations are not available for a CORBA client directly. If a server programmer wants the client explicitly handle transaction boundaries, then he/she should implement an additional IDL interface, which will contain relevant ODA calls.

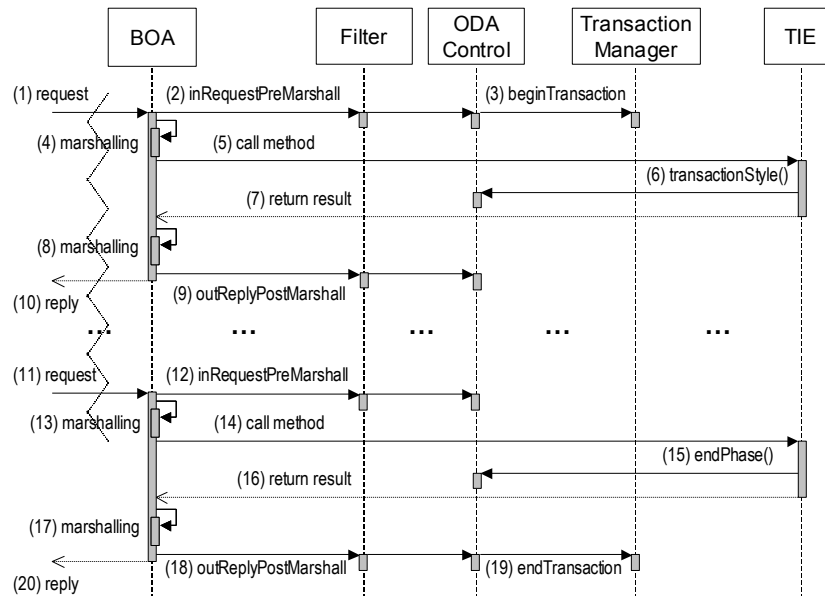


Figure 21. Phased transaction style

There is an additional requirement to the CORBA server, when the *phased* style is applied: it should be registered in the *per-client-process* mode (see Section 2.2.8). Since locking mechanisms of the most ODBMSs (e.g. ObjectStore) work on a per-process basis, ODBMS's concurrency control mechanisms will perform properly, only if different CORBA clients interact with the database using different CORBA server-side processes. Indeed, if one CORBA server process is activated and shared by different CORBA clients, the database server will fail to distinguish separate client's requests treating them as coming from one database client.

The cardinal difference between both styles is that the transaction management performs *implicitly* for the client in the first case and *explicitly* in the second. Indeed, when the *phased* style is set, responsibilities of transactions management are delegated to CORBA clients, which explicitly handle transaction boundaries. The main advantage of the *perOp* style is that it allows CORBA clients to work with persistent objects in the same way that they did with usual CORBA objects. They need to know nothing about transactions. In addition, it does not lock the database objects for a long time in the *phased* style that makes them quickly available for other concurrent clients. The disadvantage of this style is a restriction that any transaction may span no more than one operation call. For some applications, it may be inefficient to start and to end transactions so frequently. In this case, the explicit transaction control method – *phased* is preferable. The measurements made in one client environment at the server side showed that in the *phased* style the operation invocation is approximately two (read-only) – five (update) times faster than in the *perOp* style.

3.3.2.2 Drawbacks of the ODAF

Being the important step towards facilitation, regulation, and standardisation of the CORBA/ODBMS integration process, the ODAF is not devoid of deficiencies. The main shortcoming of the ODAF-made CORBA/ODBMS integration model is the inherent disability of representing several database objects by one CORBA interface. Indeed, there is a rule enforced by the general adapter pattern [GHJ+94] establishing one-to-one relations between CORBA objects (TIEs) and persistent database objects. ODAF does not give the programmer any convenient way to introduce modifications into this schema.

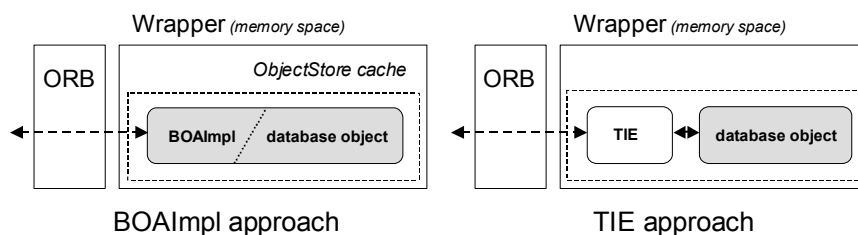


Figure 22. ODAF-supported approaches for realising persistent objects

Figure 22 presents the both approaches for realising persistent objects in ODAF. The first one is based on CORBA's BOAImpl approach. Since the modification of persistent database classes is inevitable here, this approach is not usable for the integration of existing databases and does not allow to represent several database objects by one CORBA interface. Another possibility is to use CORBA's TIE approach. The typical scenario here is that every TIE object is assigned to one concrete persistent object through reference, which is initialised during the TIE creation and serves as a bridge through which the delegations are made to the persistent object it refers (Figure 22). However, such a model is very simple and straightforward. In real-life systems, where hundreds or thousands of objects are stored in the database, this principle can have many undesirable consequences [BBB+98]. Some of them are the following.

- overloading of the server's memory caused by excessive number of TIE objects;
- performance penalty caused by the creation/deletion of TIE objects;
- overall system's inflexibility and bad scalability.

Consequently, it would be very desirable if one CORBA interface could cover several persistent objects. If there is a collection of persistent objects of the same type, one IDL interface reflecting the type of objects being collection's elements and one IDL interface embodying the collection's functionality, is a natural solution. One active TIE would then be able to represent all elements of the collection²⁰. Still the most advantageous option would be an IDL interface, which assembles methods/attributes of several persistent objects of the *arbitrary* type. In this case, the related TIE object would have to store references to every persistent object, and the choice of the appropriate reference would be made within the body of each operation. Naturally, only the subset of the original object's attributes/methods can be presented via IDL. Again, such an approach is prohibited by the policy of ODAF, which incurs every ODA-controlled TIE object map to distinct persistent (implementation) objects.

²⁰ The TIE would detect the target object and dynamically change the value of the reference to direct the upcoming request towards proper implementation object. To accomplish that, the marker mapping mechanism would have to be applied in Orbix's filter to detect whether the current request is directed to an object belonging to the collection. Such a scheme would be rather complex, non-clear, and could have substantial overhead.

In addition, there are some serious drawbacks in the ODAF transaction management. Some of them are the following.

- Transactions management is not available for the client directly. The server programmer should first make an additional IDL interface.
- ODAF proposes only two styles for transaction management – *perOp* and *phased*. As it was shown in the literature [AZ00], for real applications these styles are unpractical and suffer large performance penalties.
- It is not possible to identify transactions and therefore to nest them. Sometimes, nested transactions can significantly improve the application's performance in comparison with one-layer *phased* transactions in the ODAF model.

3.3.3 Front-End Generators

This approach puts forward the concept of a tool automatically generating the components of the middle tier layer mediating between the CORBA environment and the ODBMS.

This solution possesses various valuable advantages such as the facility to create integrated CORBA/ODBMS systems with minimum programming effort. The examples of such systems are the *Front-End Generator* (FEG) [Ami98] and the pre-processors of the ODL [SLY+99] described in Section 3.5.3. Main differences between all these systems are in the used input information and the amount of the code generated.

The *Front-End Generator* (FEG) represents a customisable code generator for a proprietary ODAF-like adapter. Based on the structure of objects stored in a pre-existing database, FEG automatically generates the code for IDL interfaces and corresponding TIE classes.

The generator incorporates various techniques aimed to enhance the flexibility and adaptability of the generated code. Through additional input parameters and files the programmer can custom the code generated. For example, the view file allows programmers to specify how particular database objects and their attributes/methods should be exposed in the CORBA environment.

However, while producing valuable results, this approach leaves some work to be completed. It also brought other open research topics on the forefront. In particular,

- The one-to-one correspondence between CORBA and database objects assumed in the FEG could be altered in order to obtain enhanced flexibility of the generated code. In the ideal case, a generator should allow programmers to build CORBA objects based on different attributes/methods of different database objects. Only in this case the full functionality of database views will be achieved.
- Some additional techniques, like „bulk“ operations described in Section 6.5, can be incorporated into the generated code.
- Further sophisticated transaction management techniques are necessary in order to improve the integrated system's performance.

3.3.4 Approaches for Transaction Management

Understanding the importance of the last issue the author of the FEG proposes a novel style for transaction management called – *preemptive* style [AZ00].

As it was described earlier, ODAF defines two styles of transactions, *per-operation* and *phased*. However, very often neither solution is satisfactory: a transaction per-operation is too costly,

yet clients should not always be aware of destination objects persistence (as it is in the *phased* style), nor manage the server's memory.

The proposed *preemptive* style utilises the fact that a single-threaded CORBA server, after receiving request from a client, usually does not accept another incoming operation request until it has completed the current one. The requests (as well as other CORBA events) are queued. A similar behaviour has a database server, which serialises subsequent transactions by not beginning the next one until the current is not committed or aborted. As a result using the *preemptive* style the ODA tries to keep the transaction open, while the subsequent requests are coming from the same client.

Actually, such a schema acts as the *phased* style, when the consecutive request comes from the same client, and acts as the *per-operation* style when the clients are different. Thus it combines advantages of both standard styles: being semantically clear (clients are separated from the server's memory management and being unaware of the target object's persistence) and at the same time not having performance overhead of opening and committing transactions on every method invocation.

The main disadvantage of this style is that transactions are processed sequentially. For example, if one client has started an operation, the other client should wait until the first operation finishes. Thus, this strategy can be a serious bottleneck in the multiclient environment. Furthermore, in fact, mixing transactions from different clients, this strategy violates ACID principles.

3.4 Persistent CORBA Services

The need for a persistence service has also been identified by the OMG consortium. Possibly, because ODMG's ODA proposal was perceived by many as biased towards object-oriented databases, and hence distant from the mainstream of CORBA's world, no further OMG specifications have contemplated the ODA approach. Instead, OMG attempted to define a service that will support object's persistence.

The first attempt was *Persistent Object Service* (POS) [OMGpos94, OMGpos00]. Unfortunately, the service was not successful because of its complexity and vagueness. Some research groups that tried to implement it have discovered many ambiguous areas in the specification [KPT96a, KPT96b, LDH99]. Two examples are the vague semantics of the operations and the weak specification of how the POS interacts with the database and other Object Services. Consequently, the POS does not yet have any complete implementation and therefore we will not consider it as a sound approach for CORBA/ODBMS integration.

As a result, some alternative solutions for a persistent service were proposed [Tuma97]. These ideas made a basis for the second version of the CORBA persistence service, which was the *Persistent State Service* (PSS) [OMGpss99]. The goals of the PSS include addressing old POS problems, while at the same time being readily applicable to existing storage technologies, and focusing on a way for CORBA objects to maintain their own persistent state. Additionally, PSS is tightly integrated with the IDL type system and the *Object Transaction Service* (OTS). PSS's close integration with OTS facilitates the development of portable applications that offer transactional access to persistent data such as a DBMS.

Overall, the PSS is a radical deviation from the POS. Rather than attempting to encapsulate all possible uses of persistence; the PSS defines an internal interface to persistence mechanisms. This section begins by highlighting the major differences between the POS and PSS specifications before the *Object By Value* (OBV) specification is discussed (the PSS is built on top of the OBV specification). A more in depth study of the PSS is then presented and

we describe how the PSS addresses the problems associated with the POS before finally highlighting the failings of the PSS.

3.4.1 Differences between the POS and the PSS

The POS and the PSS have the following major differences between.

- In the POS the persistence behaviour of an object was exposed to clients. The PSS indicates that the persistence behaviour of an object occurs totally within the domain of that object. The role clients play (if any) in the persistence process is exclusively at the candidate objects discretion, i.e. the persistence mechanisms are not visible to the clients. An implication of this is that the client cannot arbitrarily use persistence, mechanisms without first becoming a CORBA servant.
- The POS attempted to define a specification that allowed a graph of CORBA objects to synchronise their persistence operations with each other, the unit of persistence being the graph of objects. The PSS, however, takes a more pragmatic view by employing some defined state of the object itself as the unit of persistence. The defined persistence mechanisms do not operate over graphs of CORBA objects.
- PSS is tightly integrated with the IDL type system and the Object Transaction Service (OTS). PSS's close integration with OTS facilitates the development of portable applications that offer transactional access to persistent data such as a DBMS.
- The mechanism defined to move state to/from the candidate object is no longer purely abstract (as was the case with the Persistent Data Service, Protocol and Datastore abstractions of the POS). The PSS is dependent on the use of the *Persistent State Definition Language* (PSDL) specification to move state between the object and store. Due to many problems with IDL, value types in the first version of PSS, in the second version it was chosen to extend IDL with new constructs to define interfaces for storage objects.

3.4.2 Overview of the PSS

The *Persistent State Service* (PSS) is a CORBA service for building CORBA servers that access persistent data.

3.4.2.1 Basic Concepts

The Persistent State Service presents persistent information as *storage objects* stored in *storage homes*. It can only contain storage objects of a given type. The type of a storage home defines this storage object type, plus operations and keys. Storage homes are themselves stored in *datastores* (Figure 23). A datastore is an entity that manages data, for example a database, a set of files, a schema in a relational database.

In order to access a *storage object*, a transient object called „*storage object instance*“ is used. A storage object instance may be bound to a storage object in the datastore, and provide direct access to the state of this storage object: updating the instance updates the storage object in the datastore. Such a connected instance is called a storage object *incarnation*. Similar to the storage homes for storage objects, *storage home instances* are used for the aggregation of *storage object instances*. Storage home instances themselves are provided by *catalogs*.

A logical connection between the process containing instances and the datastore that contains storage objects is called *session*. A session can give access to more than one datastore. The management of sessions can be explicit or implicit where they will be managed by one or

more *session pools*. Sessions and session pools are the two kinds of catalogs defined by the PSS specification.

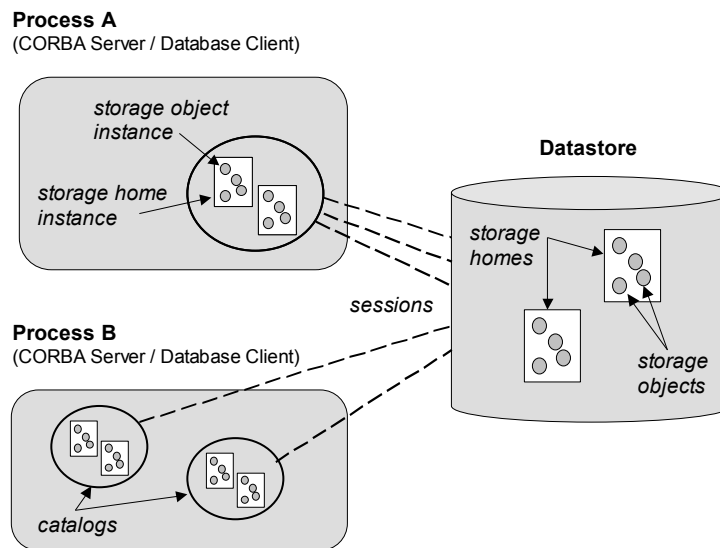


Figure 23. PSS Basic Concepts

3.4.2.2 Persistent State Definition Language (PSDL)

The PSS provides two ways to define the datastore schema and the API of storage object instances.

- Using the Persistent State Definition Language (PSDL);
- Direct in the programming language; this is known as *Transparent Persistence*.

PSDL is a superset of OMG IDL, with five new constructs: storagetype, storagehome, catalog, abstract storagetype and abstract storagehome, for example.

```
// PSDL
abstract storagetype Person {
    readonly state long social_security_number;
    state string full_name;
    state string phone_number;
};
abstract storagehome PersonHome of Person {
    Person create(in long ssn, in string full_name,
                 in string phone);
};
catalog People {
    provides PersonHome person_home;
};
```

A PSDL compiler will process this specification and generate code in the required target programming language. For example, if the target programming language is Java, the tool will generate a Java interface for each abstract storagetype, abstract storagehome and catalog. An abstract storagetype can have state members and operations. Storage objects are created and located with the help of the operations defined on the storagehome where the objects are stored (or will be stored). An abstract storagehome can define arbitrary operations.

The PSDL type model is very similar to Java: a PSDL storage type (comparable to a Java class) can implement any number of abstract storage types (comparable to Java interfaces), and it can inherit from at most one other storage type. Likewise, a PSDL storage home type

can implement any number of abstract storage homes and inherit from at most one other storage home type.

3.4.2.3 Implementing Storage Objects and Storage Homes

A PSS implementation offers several ways to define the storage types and storage home types that implement the storage object and storage home specifications specified in PSDL.

In the standard way the storage types and storage homes can be defined in a PSDL specification, using the `storagetype` and `storagehome` constructs. A compliant PSS implementation must understand these definitions and generate a full (default) implementation target programming language. For example, if your target programming language is Java, this tool will generate concrete Java classes for both `PersonImpl` and `PersonStoreImpl`.

```
// PSDL
#include <People.psd1>
storagetype PersonImpl implements Person {};
storagehome PersonStoreImpl of PersonImpl implements PersonStore {};
```

The second method is to define storage object implementations directly with regular programming language constructs, for example.

```
// Java
public class JPersonImpl implements JPerson {
    private long _ssn;
    private String _name;
    private String _phoneNumber;
    public long socialSecurityNumber() { return _ssn }
    // etc.
}
```

With transparent persistence, however, it is not possible to define application-specific storage homes: default storage homes with no keys and no operations are implicitly defined. The type hierarchy of these default storage homes parallels the type hierarchy of the corresponding storage homes. For example, `JPersonImpl`'s associated storage home type derives from `java.lang.Object`'s associated storage home type. As a result, with this method it is possible to define only a single storage home family in each datastore.

The most visible benefit is that state members may be directly represented with fields (or member variables) rather than requiring accessor and modifier methods that make calls to the PSS implementation. To provide this benefit, PSS implementations that provide transparent persistence need to make sure that an object's incarnation is loaded before the program tries to access a state member from it. It also needs to be able to determine which objects have changed and need to be committed. The approaches used to accomplish these tasks dictate some of the restrictions this standard makes on the classes that can be made persistent.

The PSS does not define a new kind of transaction: a transaction is a unit of work managed by an implementation of the Transaction Service. Transactions are created and managed through the interfaces defined by the Transaction Service. Therefore, for transactions, the proposed Persistence State Service depends on the Transaction Service. To make this dependency lighter, a concept of new „lite“ compliance level in the Transaction Service is proposed. A transaction of a lite implementation does not need to support more than one registered resource, and as a result, a lite implementation does not need to perform any coordination, logging, etc. For a more detailed description of the PSS service see the service specification [OMGpss99].

3.4.3 Drawbacks of the PSS

The PSS is designed to bridge the gap between the worlds of CORBA objects and datastore technologies. However, the worlds could hardly be further apart. On the one side, the emphasis is firmly on openness and inter-working, while on the other side, proprietary architectures prevail. This has the effect of making the object less „reusable“ on systems with different database technologies. The PSS attempts to solve this by placing an abstract layer between the CORBA object and datastore. Based on PSDL, this layer requires two extra transformations before the data is mapped into the datastore. This can significantly drop the performance of the application.

However, a PSS implementation that supports the definition of storage objects directly in Java or C++ avoids this penalty in the cost of some restrictions. A PSS implementation that provides transparent persistence for C++ must implement the ODMG version 2.0 standard for C++ ([CB99], Chapter 5), with some specific modifications that are not supported by the most database management systems. Therefore, likewise the POS, PSS also cannot support legacy objects due to candidate server being able to support the PSS specification [LDH99].

Although the specification suggests that datastore specific code could be automatically generated, tools to provide such a task would be bound to a particular datastore [LDH00]. These tools would also depend on the datastore having a well-defined schema. For example, it is hard to see how tools could be written to produce the database specific code supporting a legacy flat file system. Ad hoc development of an agreement regarding the structure of the data is necessary for all range of different applications (CORBA and non-CORBA) potentially sharing this data.

Although the specification defines how the lifetime of the persistent state is managed (using persistent storage homes), the mechanisms it specifies (datastore handle and persistent id combination) for identifying this state are abstract enough to prevent the integration of rival implementations. For example, our example implementation maintains the `database_handle` abstraction within the database specific implementation. Another compliant implementation might choose to maintain the database handle on the originating servant (imagine if some state was required to be mapped into multiple datastores).

Related to this is a more general problem of object evolution. As there are potentially multiple systems connecting to a datastore, each of these systems (and the datastore system) evolves independently. There are no semantics defined by the specification to control this evolution. Consequently, a change in the datastore will require all PSS implementations to re-evaluate the binding between datastore state and CORBA object. This is likely to involve at least the re-linking of servants using the datastore, and the regeneration of datastore specific code. The lifetime of a servant using the PSS is restricted by the evolution of the datastore.

In sum, the newly defined PSS addresses most of the problems associated with POS through simplification. Although this results in a simpler, less abstract model, it still suffers from major problems, namely:

- the specification of abstractions identifying persistent state could lead to proprietary architectures,
- there is a performance overhead in mapping state to the intermediate OBV form,
- no semantics are defined for controlling object evolution,
- there is a weak concept of equivalence (defined in the OBV specification), and
- there is no explicit support for graphs of CORBA objects.

3.5 Other CORBA Services

In the following, we summarise the two CORBA object services Query and Transaction, which are relevant to support the persistence in CORBA.

Wells and Thompson [WT94] give a critical discussion of the four original submissions to the OQS. It points out many weak points that can still be found in the final specification, such as missing meta-data services, no treatment of query optimisation or indexing, and a generally insufficient perception of queries.

3.5.1 The Object Query Service (OQS)

The *Object Query Service* (OQS) [OMGoqs00] is a simple, generic service for the distribution of queries in a CORBA environment. Objects implementing the OQS interfaces can receive queries that are formulated using defined query languages: the OQS requires the acceptance of SQL-92 or OQL [CB99] queries, at least.

Queries are represented as either strings or special query objects that support multiple executions. They are distributed using a minimal collection interface that passes a query to all their members. Each member is either again a collection, which leads to a recursive transmission of the query, or an object evaluating the query itself. Questions such as query optimisation or query decomposition are not handled by the OQS, nor does the OQS define methods for the retrieval of meta-data such as schema.

Our evaluation of these services yields very little advantage one could gain by using this, which might be a reason for the fact that vendors do not seem to have an interest in implementing this service.

3.5.2 The Object Transaction Service (OTS)

The *Object Transaction Service* (OTS) [OMGots01] provides distributed transaction processing facilities on top of CORBA, building on the notion of transactions as employed in database management systems and TP (Transaction Processing) monitors of client-server systems. The OTS defines a number of interfaces and interaction models for processing (nested) transactions and for driving the two phase commit protocol between multiple objects that may manage persistent data by different means (flat files and/or databases). One processing model addressed concerns the use of native database transaction facilities, making use of X/Open XA switches of databases (leaving responsibilities to the DBMS). Other models represent persistent data as own objects implementing specific OTS interfaces (giving responsibilities to the CORBA OTS managing component).

The OTS aims to guarantee transactional properties to all programs that run under its supervision. It handles all resource registrations, up to the coordination of any rollback or commit of a transaction. As databases typically provide transaction facilities, the OTS may well make use of these features. Database integration in CORBA for the OTS therefore is naturally implemented by using the OTS' XA interoperability features.

3.5.3 Solutions based on OTS

Instead of using *Object Database Adapter* (ODA) or persistent services, these approaches use other CORBA services to access database objects from the CORBA environment. One of them is a new integration architecture proposed by [SLY+99]. Instead of using the ODA suggested by the ODMG or CORBA's persistent services, this approach uses the combination of the *Object Transaction Service* (OTS) and wrapping techniques.

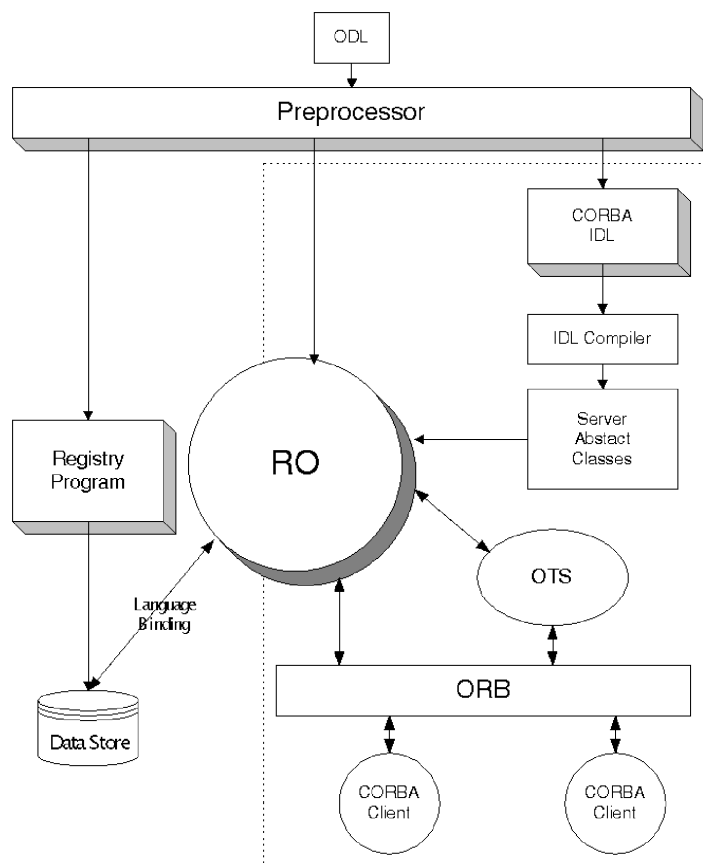


Figure 24. The CORBA/ODBMS integration architecture

Figure 24 illustrates the functional components of this CORBA/ODBMS integration architecture. The components consist of CORBA Client, OTS, RO Servers, and a pluggable ODMG-compliant ODBMS. Interoperations between these components are based on the standard CORBA and ODMG interfaces. Wrapping techniques are used to combine the CORBA transactional object with ODBMS object into the RO server object. The RO component consists of several subcomponents: RO server process, RO Factories, and RO objects. The RO server process provides RO Factories to the CORBA clients. RO objects are created by the RO Factories within the RO Server. They play the role of server object implementations and support the binding with the ODBMS.

For an ODBMS developer, in a general scenario, the programmer must define the object schema by ODL. After the preprocessor processes the ODL definition, the meta-information will be registered to the data store. The preprocessor will also generate the necessary header files and glue information. All that a developer needs is to implement the object in ODBMS way without caring about the knowledge of CORBA.

For a CORBA client application programmer, the IDL object interface, which is generated from the ODL defined by the ODBMS developer, is the only important thing he/she needs to know.

Actually, the RO component in the integration architecture plays the role of the ODA and the corresponding TIE objects. Comparing with the presented above FEG approach, where only TIE objects are generated, this approach has larger overhead due to the generation of the complete adapter. The main advantage of the presented architecture is the usage of the standard CORBA service that also supposes a smooth integration with other services.

3.6 Shortcomings of the existing CORBA/ODBMS Integration Tools

The evolution of object-oriented CORBA/ODBMS integration tools beginning with rigid and unproductive early adapters and non-generic hand-coded wrappers resulted in the emergence of generic frameworks. These approaches can be grouped according to hand-coded wrappers, adapters or adapter frameworks (such as ODAF²¹), persistent services, front-end generators and other combinations. Naturally, each technique has its own advantages and drawbacks.

Early adapters are compliant with the CORBA standard and meet OMG and ODMG philosophies. Concentrating the whole functionality in a single logical piece of code, such adapters allow an easy, fast and error-free implementation of new CORBA-compatible databases. At the same time, however, they are too rigid and do not allow smooth integration with existing databases. Storing redundant CORBA-specific information in the database, they make the integration with legacy and pre-existing data extremely difficult [BBB+98].

Wrappers, in contrast to early adapters, are fitting perfectly the situations where the integration with existing databases is required. However, their flexibility, coming at the cost of much more programming efforts, makes the implementation of integration mechanisms complex and error-prone. Detailed tradeoffs between early adapters and wrappers are thoroughly presented in the literature [VA94, Baker97, ZA97].

With the advent of *Orbix Database Adapter Framework* (ODAF) the programmers receives more options for the implementation of custom adapters. The main contribution of ODAF is a flexible architecture that can be implemented for a wide set of ODMG-compatible database management systems. However, because of some strong limitations this technique could not find a wide acceptance in the community. On the one hand, the tight integration with a proprietary ORB implementation (Orbix) prohibits the portability of ODAF-made adapters. On the other hand, generally, this technique gives not much help. ODAF only defines the outline of basic ODA components and abstract interfaces leaving their implementation to the programmer. All functionality, which normally is provided by an ODA, has to be coded „by hand“. Besides, the proposed styles for transaction management (*perOp* and *phased*) are unpractical and suffer large performance penalties [AZ00].

Further developments of ideas to CORBA/ODBMS integration represent custom pre-processors and generators. This approach puts forward the concept of a tool automatically generating the components of the middle tier layer between CORBA environment and ODBMS. An example of such a system is the *Front-End Generator* (FEG) [Ami98]. It is a customisable code generator for a proprietary ODAF-like adapter. Based on the structure of objects stored in a pre-existing database, FEG automatically generates the code for IDL interfaces and corresponding TIE classes. However, the one-to-one correspondence between CORBA and database objects assumed by the FEG prevents it is utilisation for real databases, often consisting of complex classes. To produce valuable results, the generator should be highly customized, which in the case of big projects can be cumbersome, time-consuming and error-prone. Besides, the resulting code is non-portable, lacks unified interface and strongly depends on proprietary ODAF solutions adapted to specific application requirements.

The *Persistent Object Service* (POS) [OMGpos00] is largely unimplemented due to problems identified in Section 3.3.3 of this document [KPT96a, KPT96b, LDH99]. The *Persistent State*

²¹ It should be noted, however, that ODAF is a proprietary product of IONA Tech. and does not address the issue of databases integration with other CORBA implementations than Orbix.

Service (PSS) [OMGpss99] and other similar persistent services [Tuma97] are more solid solutions that attempt to provide persistence in the form of a CORBA' service. The PSS attempts to solve the problems of POS by placing an abstract layer between the CORBA object and datastore. However, based on PSDL, this layer requires some extra transformations before the data is mapped into the datastore [LDH99]. Thus, the flexibility afforded by such a model is achieved at the cost of the performance. Although a sufficiently powerful IDL compiler could help the application developer by producing the necessary code to map state to and from the PSDL format, it is doubtful that the implementation of the PSS servants can be generated automatically. Further, as a servant evolves, the PSS servant that provides its interface to the persistent store must also evolve. This necessarily binds the application implementation code to the datastore specific code [LDH00]. Therefore, like the POS, PSS also cannot support *orthogonal persistence*.

The performance also depends on query optimisation, which is in general better if large queries are submitted to the database engine, as very few commercial DBMS are capable of inter-query optimisation. Some of these problems associated with the POS and PSS are solved in the systems using the *Object Transaction Service* (OTS) [OMGots01], which is a powerful and flexible service that is independent of the underlying database (as long as it supports the XA interface), data model or data schema. This is not true for the *Object Query Service* (OQS) [OMGoqs00]. Queries depend on the schema, and the schema depends on the data model. We think that the OQS is not addressing these problems properly. It appears to be the least common denominator of many different and incompatible database techniques, where this denominator is so small that it becomes almost useless [LTB98]. Our evaluation of these services yields very little advantage for any of them, which might be a reason to the fact why the vendors do not have an interest in implementing this service. Besides, none of the existing solutions (such as [SLY+99]) provides smooth integration of the services with legacy database applications.

From the application developer's view, superiority of one technique over another strongly depends on the requirements of the specific application. For better understanding the applicability of each approach, it is important to specify the role each system played in the particular program. The applications can be divided into two groups.

1. CORBA-based applications, where an ODBMS is used to achieve better persistence of objects, adding concurrency control, ACID transactions, fault tolerance, load balancing and other database-style functionality (described in Section 2.3.1). For these application, where no legacy persistent data exist and objects are primarily large-grained – the adapter is the right tool to use.
2. ODBMS applications, where CORBA is used to provide new software platforms/programming languages for potential clients, improve security, authorisation etc. (described in Section 2.3.2). For these applications, where pure database clients run simultaneously and a large set of fine-grained objects exists – the wrapper approach is indispensable.

Summarising, the hand-coded wrappers still are a unique solution that is usable when maximum flexibility is required. In fact, early adapters and hand-made wrappers present two extremes of the CORBA/ODBMS integration tools can have: design flexibility/implementing from scratch and design choice restriction/use of a ready product. However, a need for a tool, which would offer intermediary properties, has grown rapidly in the software community. Such a tool would take a standard-compliant form of an ODA and, at the same time, would have some degree of flexibility, adapting to specific applications requirements.

Chapter 4

Design of the eXtensible Database Adapter

The goal of this thesis is the development of a new approach providing transparent integration of Object-Oriented Database Management Systems (ODBMS) with the distributed heterogeneous CORBA environment.

This chapter presents both the design and also an overview of the approach. Describing the basic methodology in the first sections, this chapter continues with the design of the distributed CORBA/ODBMS object model, supporting framework and its major components.

4.1 Overview of the Approach

The story of the *eXtensible Database Adapter* (XDA) started in 1997 when it became clear that available approaches could not meet the requirements for the integration laid down by our research projects carried out in the context of „Interoperable Geo Information Systems“ (IOGIS)²² initiative [BBCS97]. Therefore, this thesis puts forward a new approach for CORBA/ODBMS integration, which mission is to overcome these problems.

The approach is based on a new *distributed persistent object model* that, on the one hand, simplifies the design and the implementation of new CORBA-compatible databases and, on the other hand, provides a soft, non-intrusive integration with CORBA for already existing databases. The proposed model is realised by the *eXtensible Database Adapter* (XDA) framework. Implementing new algorithms for efficient dispatching and transaction management, the framework is unique in that it permits the development of efficient and scalable CORBA/ODBMS systems dealing with large numbers of persistent objects. The XDA is a running prototype developed as part of this thesis. At the time of writing, the prototype supports the ObjectStore ODBMS [ODIug01].

4.1.1 The Distributed Persistent Object Model

The principal theme of this work was the development of a *distributed persistent object model*. The primary strength of the model is in their ability to keep the semantic of new CORBA interfaces as close as possible to their original database counterparts. Detailed description of the model is given in Section 4.2. Here we will list only the distinct properties of the model.

- **Transparent persistence.** The model extends the OMG model with concepts supporting distributed object-oriented communication with persistent database management systems. The main idea of this binding is to eliminate the gap between distributed (OMG) and database data models (ODMG) so that a programmer can use a single paradigm for designing distributed persistent objects. Thus the basic advantage of the model is a property of *transparent persistence*: there is no differences between application client code working with either distributed transient or persistent objects. The model allows the traversal and modification of the persistent data structures transparently, without explicit calls to read and write the data. Moreover, a

²² This work is funded by the German Research Foundation (DFG) within the collaborative research center SFB350 at the University of Bonn and the joint project „Interoperable geo-scientific information systems“.

remote client can not only read persistent data, but also modify and commit the modifications so that the results are permanently recorded in persistent storage.

It is a very attractive property for „pure“ CORBA clients that are completely unaware of the persistence of processed objects. The model hides the overhead imposed by the persistence and makes this feature transparent for the clients. However, the transparency requires the presence of efficient mechanisms for *automatic transaction control* and for the *management of server objects*. Moreover, for advanced clients, the model also provides the possibility of customising the behaviour of these mechanisms and *access to the native database functionality*. That is full control of the management of databases, transactions and the clustering of persistent objects.

- **A soft, non-intrusive integration.** Since a schema evolution may be a very resource intensive process for large data volumes, by realising a *mediator-based approach* the model permits the CORBA-conformance without any modifications to existing databases and applications. It is also important that databases that are remotely accessible through CORBA are at the same time available for local database applications.

On the one hand, the introduction of intermediate mediator components enhances the *flexibility* and *adaptability* of the integrated system, as well as allowing the realisation of „object views“ of the structure of persistent objects. On the other hand, however, it brings a burden for mediator management. Our own contribution here consists in identifying the important components of the programming model for *robust mediator management*, pointing out design alternatives, and collecting those components into a well-rounded execution model.

4.1.2 The eXtensible Database Adapter (XDA)

The *eXtensible Database Adapter* (XDA) framework realises the proposed distributed persistent object model. Our contribution here consists in the demonstration of the *vitality* of the proposed approach as applied within concrete sample implementations and the proposal of *new efficient algorithms* for dispatching and transaction management.

Actually, two XDA prototypes were developed. The first prototype allowed us to identify the *crucial problems* appearing in the integration process and to propose *corresponding solutions*, which were realised in the second version. The prototypes are described in Chapter 5 and Chapter 6, respectively. The prototypes were used and evaluated in several research projects [BCG+98, BCG+00, SS01]. Results from one of the projects are presented in Chapter 7 [STCK02]. The main distinctive features of the XDA framework are as follows.

- **Easy implementation of efficient and adaptable mediators.** The proposed framework *simplifies* the development of interconnecting mediator components. The framework not only gives the necessary *guidelines* for building *scalable well-performing* CORBA/ODBMS applications, but also reduces the costs of their development providing *adaptable* pre-defined *ready to use components* (Section 6.2). Some additional techniques, such as the „bulk“ operations described in Section 6.5, can be incorporated into the code. Two alternative approaches intended to simplify the development of the ODBMS-adapters are studied: *dynamic mediators* and *automated code generation* (Sections 6.2.2 and 6.2.3). The XDA can be configured for various ODMG-compatible database management systems, which is made possible by its flexible system design.
- **An efficient and flexible ODA core.** New approaches for efficient request dispatching and transaction management are proposed. Providing transparency of

persistent objects, the XDA realises efficient *automatic management* of database-specific functionalities, such as *transactions* and *clustering* (Section 6.3). At the same time, for advanced clients, the adapter provides the possibility of *customising the behaviour* of these mechanisms and access to the native database functionality, for example manually control the databases, transactions and clustering of persistent objects.

Further, to improve *performance* and *scalability* of the integrated system on databases with large numbers of persistent objects, the XDA performs *efficient dispatching* of requests to persistent objects and *intelligent caching* of corresponding mediators. A new approach for efficient *persistence-aware request dispatching* is presented in Section 6.4.

4.2 General Design Issues

The design of an integrated CORBA/ODBMS system depends on how persistent capabilities are made available (either at compile time or at runtime), and how concurrency and recovery are supported in the underlying ODBMS. Since the native interfaces of the most existing ODBMS are not CORBA-conform, the development of the integrated system is not straightforward.

The primary strength of object-oriented databases lies in their ability to model complex objects and inter-relationships among them. Therefore, it is very important to keep the semantic of their CORBA interfaces as close as possible to their original database counterparts. Since the vast majority of ODBMSs add database functionality to C⁺⁺²³ and because our projects are also focused on C⁺⁺-based tools, our attention will be primarily concentrated on those systems²⁴. The most of them, as well as ObjectStore also, provide persistence for C⁺⁺ objects using heap-style allocation/deallocation. An object is made persistent and gets its persistent capabilities when instantiated (*persistence-by-instantiation* strategy). For this purpose, the ODBMS provides its own overloaded form of operator `new()` and requires that persistent objects to be created by this operator. One could naively use the overloaded form of operator `new()` to instantiate „persistent CORBA objects“ also but, this would not work, because the CORBA’s Object Adapter by default does not know about the new overloaded operator `new()` and, therefore, can not instantiate new persistent objects.

Thus, on the one hand, a CORBA server cannot directly place CORBA objects that it implements in persistent memory. On the other hand, the other way of making CORBA-compatible persistent objects from existing database objects is also not easy. Object Adapter (which keeps a per-server-process table of active CORBA objects) can not deliver incoming requests further to database objects, since they do not have the status of CORBA objects and the adapter does not know how to access/load inactive database objects. To receive this status database objects must have corresponding IDL-conformant interfaces and be registered with the Object Adapter. Taking into account the fact that we have to deal with existing data stores that are permanently in use by multiple database applications, making such data stores CORBA-compliant should not disturb already existing applications. Since for large data volumes a schema evolution may be a very exhausting process, their CORBA-compliance in ideal case must be achieved without any changes in their database schemas.

²³ The other major alternative was Smalltalk, which support is recently being dropped by the most of ODBMS vendors in favour of Java.

²⁴ ObjectStore, ODI Inc., <http://www.odi.com/>; POET, POET Software Inc., <http://www.poet.com/>; Versant, Versant Object Technology, <http://www.versant.com/>; O2, Ardent Software, <http://www.ardentsoftware.com/>

4.2.1 Using Mediators as Intermediate Communication Components

According to the previous discussion it is clearly no way for integration of CORBA with ODMBS without some modifications in their functionality. Since our approach is oriented to third-party implementers with no access to internal interfaces of both systems, their integration is possible only by means of additional communication components [BBCS97].

In C++ databases like ObjectStore, database entities are represented by instances of C++ classes. A natural solution to make these database objects available to the CORBA clients is the utilisation of additional programmer-defined CORBA-capable objects that would mediate between them converting data parameters and delegating all function calls in the both directions (Figure 25). Because of this basic function, we will name such objects and their corresponding classes as *mediators* [SC00, Wied92]. Formally, mediator objects adapt the interfaces of existing database objects to the interfaces of corresponding CORBA skeletons/proxies, thus applying *delegation-based adapter pattern* [GHJ+94].

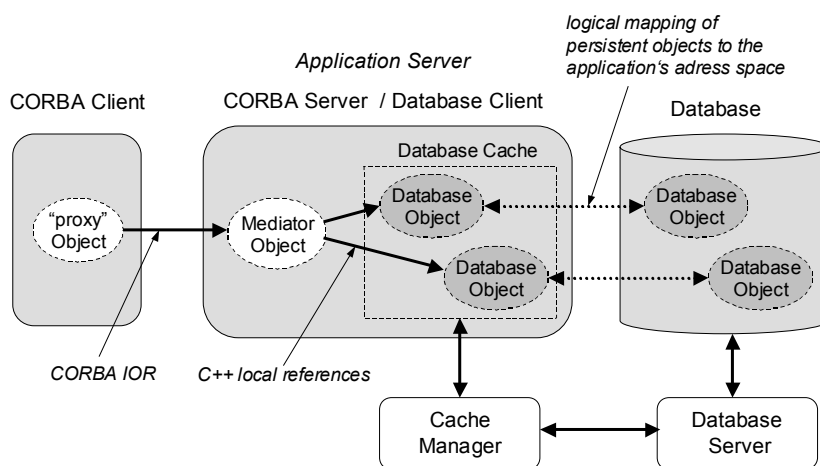


Figure 25. Mediators as intermediate communication components

Mediator classes implement CORBA-equivalents for all methods of original database classes and are, therefore, responsible for the correct data mapping between CORBA and ObjectStore C++ data models. On the one hand, providing a bridge between two systems mediator classes are responsible for the consistency of this mapping. On the other hand, they can also perform various auxiliary operations. In the literature such mediators are named as *value-added mediators* [Wied97].

Using value-added mediators, developers can give applications a runtime dynamism and self-contained intelligence that is difficult to achieve using other methods. For example, a programmer can design mediators that make compression of the transmitted data [STCK02] or protect sensitive persistent database schema from modifications and uncontrolled access. CORBA clients can access persistent database objects only through associated mediator objects. Therefore, from their secure location within the CORBA server, mediators are able to control the correctness of invocations and transmitted data.

Since mediator objects are *transient*, this fact is not only saves the database schema from undesirable modifications, but also completely separates native communication interfaces of CORBA and an underlying ODBMS. This separation can be very useful for the integration of existing databases. Deploying this approach we will not disturb existing database applications communicating with databases through native database interfaces [BBB+98].

Referential integrity of the data shared by the native and CORBA-conformant applications is guaranteed by the use of native database concurrency and recovery mechanisms.

The other advantages of mediators are the increased scalability and extensibility of the CORBA/ODBMS system. When CORBA objects are separated from database objects, it is possible to reduce the number of corresponding mediator classes and to assist extensibility by promoting the *views* on the database objects.

4.2.2 Three Tier Architecture

The introduction of mediators exposes the three-tiered nature of the CORBA/ODBMS environment and the appearance of an additional middle tier is at the same time a client of the ODBMS and a server to CORBA clients. Consider the process environment in which both systems operate. Each technology implies the presence of client and server components. Five kinds of them are presented in Figure 26 below.

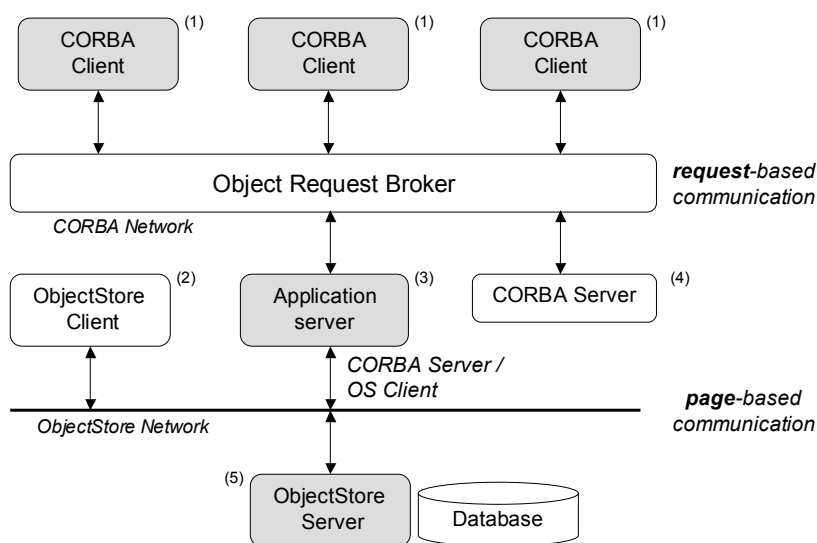


Figure 26. Three Tier Architecture of CORBA/ODBMS system

Not all of them are able to communicate with each other. Thus, to bind CORBA clients (1) with database servers (5), an additional intermediate component is required. In our architecture, we call this component an *application server* (3). The *application server* (3) is a CORBA server (like 4) to ORB clients (1), and a client (like 2) to the database server (5). To improve the performance of the communication between them, the database server and the application server can be placed into one system or merged if the DBMS allows a client and a server to share the same process.

The primary role of the application server is to provide a database-independent way of making persistent database objects available from the CORBA environment, which is achieved through *implicit maintenance* of composite mediator-based persistent CORBA objects and their object references (Figure 25). The persistence of CORBA objects also assumes the *persistence of their object references*, since „a client that has an object reference can use it at any time without warning, even if the (object) implementation has been deactivated or the (server) system has been restarted“ [OMGorb01]. Thus, driven by incoming requests, the application server should implicitly activate database objects that lie dormant in persistent memory. To allow on-demand activation of dormant objects, it must also ensure that object references handed out to CORBA clients contain information on the location of the corresponding objects in persistent memory. Hence, the application server is responsible for the *generation* and *interpretation* of references to persistent objects. Moreover, the application server offers to

CORBA clients *generic meta-interfaces* for reading or modifying data stored in the database that do not have corresponding mediators, and so provides a higher abstraction layer between clients and databases.

Other components do not differ principally from roles they play in regular CORBA or ODBMS applications. CORBA clients (1) are plain regular CORBA clients. In particular, their communication with usual CORBA servers (4) is normally not influenced by their relationship with ODBMS specific application server (3). In other words, the persistence of objects provided by the application server is *transparent* to CORBA clients.

4.2.3 Transparent Persistence

By large, the problems associated with persistent CORBA services are the result of some entity applying persistence operations to transient state. It has long been argued that persistence should not be obtained through explicit operations on an object, but should be intrinsic attribute of an object, supported by the infrastructure within which object operates [ABC+93]. That is, the persistence of an object should be *orthogonal* to *use*, *type* and *identification* of the object. In defining *orthogonal persistence* Atkinson and Morrison [AM95] cite three design principles that are desirable in any persistent programming language design, enabling the full power of the persistence abstraction:

Type Orthogonality: persistence is available for all data irrespective of type.

Transitive Persistence: the lifetime of all objects is determined by reachability from a designated set of root objects.

Persistence Independence: it is indistinguishable whether code is operating on short-lived or long-lived data.

The proposed XDA model supports *transparent persistence* that is a loose version of the *orthogonal persistence* [JA98]. The XDA application server provides an abstraction of persistent data storage that fulfils the first and the third requirements. That is, from the client's perspective access to persistent objects is *transparent*, it is independent of the *type* and *location* of the object and it is *no difference in code* operating with transient and persistent data. However, supporting the first and the third requirements the model violates the second, since the *persistence by instantiation* approach is used. This strategy does not support the property of *transitive persistence*, while once a persistent object is created, its lifetime status cannot be changed.

4.2.3.1 Practicalities

Complete persistence independence typically can not be achieved, and even if it can, it may not be desirable, since one usually wants to offer a degree of control to the programmer. For example, in using a transaction mechanism one must generally specify at least the placement of transaction boundaries (begin/end). Nevertheless, a language design would not be transparent if it is required different expression for the usual manipulation of persistent and non-persistent objects; i.e. for operations such as method invocation, field access, parameter passing, etc.

Similarly, perfect type orthogonality may not be achievable and may not even be desirable. For example, some data structures refer to strictly transient entities (e.g., open file channels or network sockets) whose saving to persistent storage is not even meaningful (they cannot generally be recovered after a crash or system shutdown). In many languages these objects are not entirely first class, and supporting persistence for them may also be challenging to implement. Thus, perfect type orthogonality, in the sense that any instance of any type can persist, is not so desirable as that any instance of any type that needs to persist can persist.

Real orthogonality can be reached only when *persistence by reachability* strategy is used: an object is made persistent when it is reachable from another persistent object. This strategy offers maximal flexibility since objects can change arbitrary their lifetime status at runtime. However, such flexibility often comes at cost of performance of the system [HC99]. The principle of persistence designation means that any allocated instance of a type is potentially persistent, so that programmers are not required to indicate persistence at object allocation time. Languages in which the extent of an object can differ from its scope usually allocate objects on a heap, where they are retained as long as necessary. Deallocation of an object may be performed explicitly by the programmer, or automatically by the system when it detects that there are no outstanding references to the object. This can be determined by a *garbage collector* [Jon96] by computing the transitive closure of all objects reachable (by following references) from some set of system roots. In systems that support garbage collection, persistence designation is most naturally determined by reachability from some set of known persistent roots. From this point of view *transient persistence* based on *persistence by instantiation* is a reasonable compromise between flexibility and performance.

4.2.3.2 Advantages of Transparent Persistence

The advantages that accrue through *transient persistence* to the design of distributed systems are many. *Persistence independence* allows programmers to focus on the important problem of writing correct code, regardless of the longevity of the data that code manipulates. Moreover, the client CORBA code will function equally well for both transient and persistent data.

Data type orthogonality allows full use of data abstraction throughout an application, since a type can be applied in any programming context. This permits the development of programming systems based on rich libraries of useful abstract types that can be applied to data of all lifetimes.

In sum, the proposed model simplifies the development of distributed persistent objects and allows client programmers to write code independently of the persistence (or potential persistence) of the data that they manipulate. Providing transparent persistence the model promotes the programming virtues of modularity and abstraction; both are crucial to the construction of large distributed persistent applications.

4.3 The Distributed Persistent Object Model

The *distributed persistent object model* is an execution model for distributed object-oriented applications using XDA. The model extends the OMG model with concepts supporting distributed object-oriented communications with ODMG-based persistent object-oriented systems. The main idea of this combination is the elimination of a gap between distributed (OMG) and database data models (ODMG), so that the programmer can use a single paradigm for designing distributed persistent objects.

This first Section 4.3.1 defines an abstract model providing an organised presentation of concepts and terminology. The design of the model depend on how the persistent capabilities are made available (either at compile time or at runtime), and how concurrency and recovery are supported. Therefore, the second Section 4.3.2 defines a mapping determining the representation of persistent C++ into CORBA's Interface Definition Language (IDL). The mapping defines a set of rules determining how interfaces of existing database C++ classes²⁵ can be represented in IDL and how these equivalent IDL structures can be implemented with the help of their corresponding C++ analogues. The last

²⁵ For short, the expression „classes whose instances are stored in database“ will be referred to as „database classes“.

Section 4.3.3 defines the concepts associated with implementation of the defined abstract model according to the mapping of persistent C⁺⁺ into CORBA's IDL.

4.3.1 Abstract Model

The presented object model is a superset of the OMG object model, which enables transparent interaction and representation of databases in the CORBA environment. Therefore, the object model is most specific and prescriptive in defining concepts meaningful to CORBA clients, including such concepts as object creation and identity, requests and operations, types and signatures.

It specifies the following basic modelling primitives.

- The basic modelling primitives are the *object* and the *representation*. Only an object has a unique identifier.
- Objects and representations can be categorised into *types*. All objects of a given type exhibit common behaviour and state. A type is itself an object.
- Behaviour is defined by a set of *operations* that can be performed on or by the object.
- State is defined by the values on object carries for a set of *properties*. A property may be either an *attribute* of an object or a *relationship* between the object and one or more other objects.
- A *database* stores objects, enabling them to be shared by multiple users and applications. A database is based on a local *schema* that is defined in the persistent C⁺⁺ language. The database contains instances of the persistent types defined by the schema.

4.3.1.1 Requests

Clients access objects by issuing *requests*. The term request is broadly used to refer to the entire sequence of causally related events that transpires between a client initiating it and the last event causally associated with that initiation.

The information associated with a request consists of an *identity for a target object*, the *operation's identity*, zero or more (actual) *parameters*, and an optional *request context*. A request may have parameters that are used to pass data to the target object; it may also have a request context that provides additional information about the request. A request context is a mapping from strings to strings.

A request causes an object to be performed on behalf of the client. One possible outcome of performing an object is returning to the client the *results*, if any, defined for the request. If an abnormal condition occurs during the request's execution, an *exception* is returned. The exception may carry additional return parameters particular to that exception.

4.3.1.2 Interfaces, Types and Classes

An *interface* is a description of a named collection of possible operations that a client may request through that interface.

Interface inheritance provides the composition mechanism for permitting an interface to aggregate (support) functionality defined on several interfaces. Therefore, all the attributes, relationships, and operations defined on a super-interface are inherited by the sub-interface. However, redefinition of inherited properties and operations is not allowed. The sub-interface may only define additional properties and operations, and cannot redefine signatures of the inherited properties and operations.

An interface defines a set of methods but does not implement them. An *implementation* is a code that implements an interface. An implementation defines data structures, exceptions, and methods that operate on the data structures to support the required persistent state and behaviour.

A *type* is an identifiable entity that defines an association between only one *implementation* (that can be abstract) and one or more *interfaces* that the implementation implements.²⁶ A combination of an implementation and a particular interface is a *class*. Thus, the type is an abstract concept and the class is an implementation concept. A class that has an interface without implementation is *abstract*.

4.3.1.3 Objects

An object system includes entities known as objects. An object is an identifiable, encapsulated entity that can be requested by a client through one or more interfaces.

All objects have the IDL interface `Object`, which is implicitly inherited by the definitions of all user-defined objects.

Objects are created using `create_object()` method of the corresponding factory interface. It can be another object, a database, a segment or a cluster.

The lifetime of an object is orthogonal to its class. This means that persistence is independent of type. The lifetime is specified when the object is created and may be.

- **Transient:** the object's memory is allocated and deallocated by the programming language's runtime system. Typically, allocation will be made dynamically in the heap.
- **Persistent:** the object's storage is managed by the database.

Each object has a *unique identity* – the object identifier, which does not change and is not reused when the object is deleted. In addition, an object may also be given one or more names that are meaningful to the user, provided each name identifies a single object within the scope of the definition of the name, that is, within a database.

Objects can have also *attributes*, *operations* and *exceptions*.

Attributes. An attribute is defined on a single object type. An attribute is not an object and does not have an object identifier, but takes as its value a literal or an object identifier. In the last case, such attributes can define relationships between types.

Operations. The instances of an object type have behaviour that is specified as a set of operations or *operation signatures*. The *signature* specifies the name of the operation, the names and the types of each argument, the names of any exceptions that can be raised, and the types of the values returned, if any. Overloading operations names are not supported.

Exceptions. The persistent object model supports dynamically nested exception handlers. As we have already noted, operations can raise exceptions and exceptions can communicate exception results. Exceptions can form a generalization-specialisation hierarchy, with the root type `Exception` provided by the CORBA.

4.3.1.4 Object Identity

A key part of the object's definition is unique identity. In our distributed object-oriented system, each object is assigned an *Object Identifier* (Object ID) when it is created.

²⁶ However, the interface definition is a specification that defines only abstract behaviour that can be realised in several implementations. This case is not allowed in the presented model, since a client can manipulate only with interfaces that should uniquely identify types.

An Object ID has following properties, it is

- System-generated.
- Unique.
- Invariant, in the sense that it cannot be altered during its lifetime. Once the object is created, this Object ID will not be reused for any other object, even after the object has been deleted.
- Independent of the values of objects (its attributes). Two objects can have the same state, but different identities.
- Invisible to the user (ideally).

An object identity is a value that reliably denotes a particular object. Specifically, an object identity will identify the same object each time the reference is used in a request. Thus, object identity ensures that an object can always be uniquely identified, thereby automatically providing entity integrity. In fact, as such an object identity ensures uniqueness network-wide, it provides a stronger constraint than the original database identity.

An object may be denoted by multiple, distinct object identities. For example, in the case if an object has several interfaces. All object identities satisfy the basic substitution principle. If interface A is derived from interface B, then an identity of an object that supports interface A can be used where the formal type of a parameter is declared B.

In addition, objects can contain or refer to other objects using object identity. However, for each referenced Object ID in the system there should be always an object that corresponds to the Object ID. Dangling references are not allowed.

4.3.1.5 Representations

A *representation* is anything that may be a legitimate (actual) parameter in a request. More particularly, a representation is an instance of an OMG IDL data type. There are non-object values, as well as values that reference objects.

A *representation type* is an entity, which shares many of the characteristics of interfaces and structs. Every type is associated with one or more representations, which instances represent the serialised state of the type's instances. A representation is a description of both a set of operations that a client may request and of state that is accessible to a client. Instances of a value type are always local concrete implementations in some programming language.

4.3.1.6 Collections

In the persistent object model, a collection contains an arbitrary number of unnamed homogeneous elements, each of which can be an instance of an atomic type or another collection. Iteration over a collection is achieved by using an iterator that maintains a current position within the given collection. There are ordered and unordered collections. Ordered collections must be traversed first to last, or vice versa; unordered collections have no fixed order of iteration.

The stability of the iterator determines whether iteration is safe from changes made to the collection during the iteration. An iterator object has methods to position the iterator pointer at the first record, get the current element, and increment the iterator to the next element, among others. The model specifies five built-in collection subtypes with fixed properties.

- *Set*: unordered collections that do not allow duplicates.
- *Bag*: unordered collections that do allow duplicates.

- *List*: ordered collections that allow duplicates.
- *Dictionary*: unordered set of key-value pairs with no duplicate keys.

Each subtype has operations to insert an element into the collection and the usual set operations; union, intersection, difference, etc.

4.3.1.7 Queries

We consider queries as operations under collections of objects, restricted to projections, selections, joins and aggregations (excluding, for instance, recursive queries) [ODIug01]. Query properties that need to be considered when integrating a DBMS in CORBA are.

- The input and the target schema are not necessarily the same. A query can be a restructuring query, i.e. the query produces a target schema, which is not a subpart of the input schema. Restructuring queries are commonplace in existing database management systems, for instance for the definition of views.
- The result of a query is in general not identical to an object of the underlying database. This raises difficult questions: What is the type of the result? Where is this type placed inside the type inheritance tree? If the query combines data from different objects, what is the object identity of the result?
- Ad-hoc queries require knowledge of the database schema. Interfaces allowing ad-hoc queries are only useful if they also provide mechanisms to retrieve the schema (meta-data).
- The actual performance of queries in languages such as OQL highly depends on query optimisation.

4.3.1.8 Metaobject Interface and Metadata

The metadata is „the data about data“. That is, data that describes objects in the system, such as classes, attributes, and methods. Many existing ODBMSs provide meta information in the object-oriented form of classes and objects, so a user can query this information as he or she can query other objects. Therefore, the persistent object model defines a set of IDL operations giving the remote CORBA user to use these ODBMSs features. It provides metadata for objects, types, attributes and operations.

4.3.1.9 Transactions

The persistent object model supports the concept of transactions, which are logical units of work that take the database from one consistent state to another. The model assumes a linear sequence of transactions executing within a thread of control. Concurrency is based on standard read/write locks in a pessimistic concurrency control protocol. All access, creation, modification, and deletion of persistent objects must be performed within a transaction, as well as a checkpoint operation. A checkpoint commits all modified objects in the database without releasing any locks before continuing the transaction. The model precludes XA-compliant distributed transaction support provided by the ODBMS.

4.3.1.10 Databases

The persistent object model supports the concept of databases as storage areas for persistent objects of a given set of types. A database has a schema that is a set of type definitions. Each database is an instance of the type Database. Named objects are entry points to the database, with the name bound to an object.

4.3.2 Mapping of persistent C⁺⁺ to IDL

The key component determining the representation of the abstract model is the mapping of persistent C⁺⁺ into CORBA's *Interface Definition Language* (IDL). Obviously, IDL and a programming language like C⁺⁺ have some essential inherent differences. The both have basic object-oriented constructions for defining the types, their attributes and methods, as well as the types of arguments and return values of the methods. However, designed only for the specification of the object's interfaces, IDL lacks any details concerning implementation. Independency of any concrete programming language allows IDL to be translated into a wide range of object-oriented as well as non-object-oriented programming languages.

In particular, IDL lacks notions of constructors, destructors, overloading of member functions, private/protected members, static members, global functions, etc. In the contrast to IDL, definitions of C⁺⁺ classes bear plenty of various implementation details and, in fact, do not separate the interface of the class from its implementation. C⁺⁺ is much richer and can be viewed as a superset of IDL [Str94]. Consequently, in general it is not possible to translate one-to-one all C⁺⁺ constructs into equivalent IDL constructs. Hence, developing the C⁺⁺ to IDL mapping efforts should be done to keep the logic and layouts of original C⁺⁺ structures during the transformation to IDL. It should be noted that since an unambiguous one-to-one mapping between C⁺⁺ and IDL is not possible, this mapping is *not* symmetrical. For example, if a C⁺⁺ class mapped to the IDL interface and then mapped back to C⁺⁺, then the resulted class will be not the same as the original.

General principles of C⁺⁺ to IDL mapping are presented below. In some cases, the mapping rules for objects' attributes, operations' parameters and return values are different allowing the same C⁺⁺ type to be mapped differently depending on whether it refers to an attribute, operation's parameter or return value. In the further description for each of those cases, when needed, the C⁺⁺ to IDL mapping rules will be defined separately.

4.3.2.1 Mapping for Names

In general, the presented mapping uses underscore as a delimiter character in constructed IDL names to separate added prefixes and suffixes from original C⁺⁺ names. Therefore, if the C⁺⁺ name has a leading underscore, the underscore is cut off (Table 3). If, because of the cut, there is a name clash with another name occurs, a unique integer is appended to the name of the method.

C ⁺⁺	IDL
<u>name</u>	name1 or name
name_	name2
variableName	variablename1
VariableName	variablename2
operationName	<i>prefix_operationname_suffix</i>
operationName	
OperationName	
typeName	XDA_ typename1 or XDA_ typename
TypeName	XDA_ typename2

Table 3. C⁺⁺ to IDL mapping for names

C⁺⁺ supports case-sensitive names and IDL does not, therefore, two variables in C⁺⁺ whose names differ only in case must be modified during the mapping to IDL. Such C⁺⁺ names are mapped to IDL by addition of an integer starting from 1 appended at the end of the original name. If there is a name conflict with another name after appending the integer, a unique integer to be appended should be found.

Another strategy for modifications of names is used for names of operations. They are distinguished in IDL with the help of extra prefixes and suffixes. The global-scope names, such as the names of constructed types and classes, are mapped to IDL by addition to the original C++ name of a prefix for the developed adapter, for example „XDA_“. If there is a name clash with another type name occurs because of the case, a unique integer is appended.

4.3.2.2 Mapping for Namespaces

C++ namespaces are mapped to IDL modules. The contents of an IDL module appear inside the corresponding C++ namespace, so the scoping of an IDL definition is preserved at the C++ level. Here is an example (Table 4).

C++	IDL
<pre>namespace Outer { // ... namespace Inner { // ... }; };</pre>	<pre>module Outer { // ... module Inner { // ... } }</pre>

Table 4. C++ to IDL mapping of namespaces

4.3.2.3 Mapping for basic Types

Most of C++ basic types have similar types in IDL. Therefore, their mapping is natural and straightforward (Table 5). In cases where there is not direct mapping, the IDL type which standard-defined mapping to C++ type encloses the given C++ type was chosen, e.g. `int` is mapped to `long`. IDL does not have a type `int`, so there are no guessing games as to its range. An IDL `short` is mapped to at least a 2-byte type, and IDL `long` is mapped to at least a 4-byte type.

The presented table does not include pointers, because their mapping is quite different from the mapping of values. An exception here was done only for a type `char*`, which as the mostly used type is mapped to a separate IDL type `string`. In general, IDL uses object references to achieve what in a non-object-oriented environment would normally be done with a pointer. However, object references can denote only objects but cannot point to data. IDL supports recursive data types, such as trees, without introducing a data pointer type (see next section).

C++ type	IDL type	IDL2C++	Size
<code>bool</code>	<code>boolean</code>	<code>CORBA::Boolean</code>	Unspecified
<code>unsigned char</code>	<code>char</code>	<code>CORBA::Char</code>	≥ 8 bits
<code>char</code>	<code>char</code>	<code>CORBA::Char</code>	≥ 8 bits
<code>char*</code>	<code>string</code>	<code>CORBA::String</code>	Variable
<code>unsigned short</code>	<code>unsigned short</code>	<code>CORBA::UShort</code>	≥ 16 bits
<code>short</code>	<code>short</code>	<code>CORBA::Short</code>	≥ 16 bits
<code>int</code>	<code>long</code>	<code>CORBA::Long</code>	≥ 32 bits
<code>unsigned int</code>	<code>unsigned long</code>	<code>CORBA::ULong</code>	≥ 32 bits
<code>long</code>	<code>long</code>	<code>CORBA::Long</code>	≥ 32 bits
<code>unsigned long</code>	<code>unsigned long</code>	<code>CORBA::ULong</code>	≥ 32 bits
<code>float</code>	<code>float</code>	<code>CORBA::Float</code>	≥ 32 bits
<code>double</code>	<code>double</code>	<code>CORBA::Double</code>	≥ 64 bits
<code>long double</code>	<code>double</code>	<code>CORBA::Double</code>	≥ 64 bits

Table 5. C++ to IDL mapping for basic types

File-scope C++ constants are mapped to the global IDL constants, and static class-scope C++ constants map to the IDL constants nested inside an interface, for example (Table 6).

C++	IDL
<pre>const long MAX_ENTRIES = 10; class Foo { public: ... static const long MAX_NAMES; // or in standard C++ static const long MAX_NAMES = 20; };</pre>	<pre>const long MAX_ENTRIES = 10; interface XDA_Foo { ... const long MAX_NAMES = 20; ... };</pre>

Table 6. C++ to IDL mapping for constants

4.3.2.4 Mapping for constructed and user-defined Types

In addition to providing the built-in basic types, IDL permits defining of complex types: enumerations, structures, unions, sequences and arrays (Table 7). It is also possible to use typedef to name a type explicitly.

C++	IDL
<pre>short tab[10]; short * tab;</pre>	<pre>short XDA_tab[10]; short tab[10]; sequence<short> XDA_tab; sequence<short> tab;</pre>
<pre>struct address { char* city; char street[20]; unsigned short no; short tab[10]; };</pre>	<pre>struct XDA_address { string city; string<20> street; unsigned short no; short tab[10]; };</pre>
<pre>union model_name { char element1; short element2; long element3; };</pre>	<pre>union XDA_model_name switch (short) { case 1: char element1; case 2: short element2; case 3: long element3; };</pre>
<pre>enum colors {red, green, blue}; class Foo { public: ... enum prc {a_vista, credit}; };</pre>	<pre>enum XDA_colors {XDA_red, XDA_green, XDA_blue}; interface XDA_Foo { ... enum prc {a_vista, credit}; };</pre>

Table 7. C++ to IDL mapping for constructed and user-defined types

C++ arrays are mapped to IDL arrays with the same length and the type of elements corresponding to presented C++ to IDL mapping. If the C++ array is multidimensional, the type of IDL array is another array and so on. If the array is represented by the pointer, then it is mapped in IDL to the unbounded sequence. Exception: the array of characters is mapped to bounded string in IDL.

C++ structs are mapped to IDL structs with the same name extended by addition of a prefix for the developed adapter, e.g. „XDA_“.

IDL unions differ quite a bit from their C++ counterparts. In particular, they must be discriminated; they allow multiple case labels for a single union member; and they support an optional default case. However, their semantics is the same as in C++ and only one member of the union is active at a time. The discriminator that indicates which member is currently active can be of any integral type (char, integer, boolean or enumerator). As in C++, IDL unions create a namespace, so union member need be unique only within the enclosing union and modification of their names by a prefix is not necessary.

Enumerated types are mapped to the IDL enumeration with the same name and names of elements if the type definition is embedded within a class' scope. If the scope of the type is global, a prefix for the developed adapter (e.g. „XDA_“) is added to the beginning of the type/elements names to avoid name conflicts between the original C++ code and the code generated by IDL compiler.

4.3.2.5 Mapping for Classes

Mapping rules for all basic C++ constructs concerning the classes are presented in the following Table 8. C++ class maps to IDL interface. The name of IDL interface will be achieved from the original C++ class name by addition of a prefix for the developed adapter, e.g. „XDA_“.

C++	IDL
class	interface
inheritance templates	inheritance <i>modification of the interface's name</i>
attributes public private and protected constant	attributes attributes with the same name - readonly attributes
operations public private and protected constant overloaded operators	operations operations with the same name - - <i>several operations with modified names</i> <i>several operations with modified names</i>
operation's parameters value reference pointer constant return	operation parameters <i>in</i> <i>inout</i> <i>in</i> or <i>inout</i> depending on the type <i>in</i> return
constructors, destructors	<i>mapped on special separate interfaces</i>
static members	<i>mapped on special separate interfaces</i>

Table 8. C++ to IDL mapping for classes

Public inheritance is mapped to IDL interfaces inheritance. Private and protected inheritance relations are ignored. Pure virtual classes and methods are mapped in the same way as usual

classes. They differ only in their implementation. Multiple inheritance on database classes is mapped to multiple inheritance on IDL interfaces²⁷. Friendship declarations are not mapped.

Instantiations of template classes are mapped to separate IDL interfaces with different names consisting of the template class' name and template arguments' names connected by underscores. An example of the mapping for template classes is given below.

```
// C++
template class my_Reference<T> : Reference
{...};
my_Reference<Person> p;

// IDL
interface XDA_my_Reference_Person : XDA_Reference
{...};
```

All public member attributes in a C++ class are mapped to IDL attributes with the same name. Private and protected attributes are not mapped. Constant attributes are mapped to readonly IDL attributes.

All public member operations defined in a C++ class are mapped to an operation definition in IDL with the same name. The exceptions are constructors, destructors and static methods. They will be discussed later in this chapter. Private and protected methods are not mapped.

4.3.2.6 Mapping for overloaded Operations and Operators

Unfortunately, IDL does not allow overloading operations. Therefore, an overloaded C++ method must be mapped to several IDL operations with different names constructed from the name of the original C++ method by addition of a postfix that is the name of the type for the first parameter. Overloaded operations, which differ only by a „const“ clause in the first parameter, will differ by an additional postfix „_const“ and a kind of the first parameter „in“ or „inout“. Unlike C++, IDL does not distinguish between operations for read and write access. Therefore, there is no difference in mapping of constant C++ operations to IDL. For example, an overloaded operation `intersect()` will be mapped to the following three IDL operations (Table 9).

C++	<pre>gtObj * intersect (gtLine * obj); gtObj * intersect (gtPoint * obj); gtObj * intersect (gtPlane * obj) const; gtObj * intersect (const gtPlane * obj);</pre>
IDL	<pre>GTA_gtObj * intersect_gtLine (inout GTA_gtLine obj); GTA_gtObj * intersect_gtPoint (inout GTA_gtPoint obj); GTA_gtObj * intersect_gtPlane (inout GTA_gtPlane obj); GTA_gtObj * intersect_gtPlane const (in GTA_gtPlane obj);</pre>

Table 9. C++ to IDL mapping for overloaded operations

Like overloaded operations IDL also does not support operators. Therefore, C++ operators are mapped to the sets of equivalent IDL operations with different names consisting of a prefix „operator_“, a literalised name of operation's symbol, plus a postfix if the operator was overloaded. Overloaded operators are treated in the same way as regular methods are. For example, an overloaded operator `operator=()` will be mapped to the following IDL operations (Table 10).

²⁷ If one class multiply inherits from two classes having a member with the same name, the resulting name clash in IDL interface must be eliminated by the programmer.

C++	void operator= (gtLine * obj); void operator= (gtPoint * obj); void operator += (float arg);
IDL	void operator_set_gtLine (inout GTA_gtLine obj); void operator_set_gtPoint (inout GTA_gtPoint obj); void operator_plus assign(in float arg);

Table 10. C++ to IDL mapping for operators

Since C++ does not have the concept of *out* parameters, except for the return value, only *in* and *inout* parameters in IDL are used. A parameter passed through a non-constant pointer or a non-constant reference is mapped to *inout* parameter in IDL operations. All other types of parameters are mapped to *in* parameters in IDL operations.

4.3.2.7 Mapping for Constructors and Destructors

The mapping conventions for constructors and destructors are different from the basic mapping conventions for ordinary functions.

Constructors are accessible through special functions `create_object()` and `create_par_object()` available on separate interfaces `XDA_Database`, `XDA_DbSegment`, `XDA_DbCluster` and `XDA_Object`.

Default constructors are mapped with the help of a function -

```
// IDL
XDA_Object create_object(in string type)
    raises (XDA_TypeNotFound);
```

Parameterised constructors are mapped with the help of a function -

```
// IDL
XDA_Object create_par_object(in string type, in any par)
    raises (XDA_TypeNotFound, XDA_BadValue);
```

The difference between the both functions is that the first one takes only one argument (the name of the type) and is implemented with the help of the default constructor, where the second takes additional parameters represented through a CORBA's *any* and can be implemented with the help of custom constructors.

The return value is set to the newly created object's reference. Here it is assumed that IDL interfaces for all user-defined mediators inherit from `XDA_Object`.

The functions are available to the client on several IDL interfaces: `XDA_Database`, `XDA_DbSegment`, `XDA_DbCluster` and `XDA_Object`. Selecting a particular interface the client can control where the new object will be created – in a database, in a segment, in a cluster or as close as possible to an existing object. For more details about these operations please have a look in Section 5.5 about XDA' CORBA interface.

Destructors are mapped to a function `destroy()` on the interface `XDA_Object`.

```
// IDL
interface XDA_Object {
    ...
    void destroy ();
    ...
}
```

These rules are made in accordance with a CORBA's philosophy of object factories. Indeed, interfaces `XDA_Database`, `XDA_DbSegment`, `XDA_DbCluster` and `XDA_Object` can be

recognised as object factories. However, in the contrast to the general philosophy of factories, the destructor is mapped on the object's interface and NOT on the factory's interface.

This solution has two advantages: (1) the client does not have to remember which factory has created an object in order to delete it; (2) the process of object deletion is more straightforward and clear since there is no need to submit a reference to object as an argument to the factory's delete operation.

It should be noted that objects' constructors and destructors play special role in ODBMSs since their invocation causes the particular object to be stored and removed from the database. Therefore, the exposition of these operations to CORBA clients would let them to add/remove data from the database, which in some cases can be undesirable.

An example of the creation and deletion of persistent objects with the help of the XDA is given below.

```
// C++
XDA_Database_var dbVar = ...;

// create an object
XDA_Object_var objVar = dbVar->create_object("GTA_Point");
// downcast to the required type
GTA_Point_var pointVar = GTA_Point::_narrow (objVar);

// or
XDA_Object_var obj2Var = objVar->create_object("GTA_Point");
// downcast to the required type
GTA_Point_var point2Var = GTA_Point::_narrow (obj2Var);

... // use pointVar, point2Var

// destroy the created object
pointVar->destroy();
```

4.3.2.8 Mapping for static Class Members

Static class members do not fit into CORBA's philosophy. Strictly speaking, they are compliant neither with CORBA model, nor with the general object-oriented model. From the other hand, they cannot be skipped in the mapping, because their omission can incur irreversible losses in application's logic. Since static class members can be accessed before an object of this class is created, it is senseless to place the corresponding IDL method in the same object's interface. In fact, static members have global scope; the classes provide only scoping for their names.

Therefore, it was decided to place all static members into a special IDL interface `XDA_Globals`. To avoid name clashes, the name of the class, to which particular static member belongs, is appended to the name of the member. The `XDA_Globals` interface is implemented on a single CORBA object that is available to the client through a corresponding function on the IDL interface `XDA_Adapter` (Table 11).

This approach has some remarkable advantages. First, there are less CORBA interfaces that should be exposed to the client. Second, it does not have to use additional factory objects for object creating that minimise an overall number of calls made through the net. Finally, the management of implementation classes on the server side is also greatly simplified – instead of creating/deleting numerous factory objects, CORBA server tackles with only one common object.

C++	IDL
<pre>class foo { public: float op(char* str); static int static_op(void); // ... static void * static_attr; };</pre>	<pre>interface XDA_Object { // ... void destroy(); }; interface XDA_foo : XDA_Object { float op(in string str); }; XDA_Globals { long XDA_foo_static_op(); any XDA_foo_static_attr(); void XDA_foo_static_attr (in any attr); };</pre>

Table 11. C++ to IDL mapping for static class members

An exception from this rule is operator `new()` that is also a kind of a static member function, but, it is mapped in the other way (see Section 4.3.2.7). It should be also noted that it is possible to place all constructors in this global common interface too, effectively eliminating the need for special factory interfaces by the cost of additional parameters for indicating where the object should be created.

4.3.2.9 Mapping for Pointers and References

It is probably the most difficult and important point in the mapping. It defines how parameters, which are usually represented by pointers and references, are passed: *by reference* or *by value*. The main problem occurs in defining a mechanism for passing *objects by value*. The single IDL abstraction that can be used for representation of objects is the interface. According to the general rules of IDL basic, template and constructed IDL types are always passed by value EXCEPT the interface type, which is always passed by reference. This makes the interface type different to all the others and applies whether the operation call is made on an object that is remote or co-located.

Copy semantics are used in the by value passing of an IDL parameter. An operation gives any value that is passed back to the client code making the call. If an interface type is passed through an operation call, then the client receives a reference to a local „proxy“ object. This transparency of the target object location is a powerful feature of CORBA. Unfortunately, it can be a strong restriction. There are situations when it would be preferable to pass the object itself. Nevertheless, how *Object by Value* (ObV) can be achieved now?

4.3.2.10 Object by Reference

C++ functions returning pointers/references to constant variables cannot be properly mapped to IDL and it can be a potential source of trouble. Generally, pointers and references to objects are mapped to IDL object references. Constant member variables are mapped to readonly IDL attributes. Non-constant arguments of methods are mapped to inout arguments of IDL methods (Table 12). *Exceptions: the pointer to character is mapped to unbounded string in IDL; member variables being reference to type are mapped to readonly attributes of corresponding IDL types²⁸; pointer to void is mapped to IDL any type.*

²⁸ This is caused by the fact, that once set (in this case typically in constructor) the value of a reference cannot be changed.

C++	IDL
<pre> Class foo; class factory { public: foo* obj_ptr; foo& obj_ref; foo* newObject (const char * name); foo& getObject (long number); }; </pre>	<pre> intefrace XDA_foo; interface XDA_factory { attribute XDA_foo obj_ptr; readonly attribute XDA_foo obj_ref; XDA_foo newObject (in string name); XDA_foo getObject (in long number); }; </pre>

Table 12. C++ to IDL mapping for pointers passed by reference

The Problem with Returning Pointers/References to constant Variables

IDL lacks semantics of returned constant variable that sometimes can follow to unexpected program behaviour. In most cases, where data is normally returned by value (mapping by value) it is not a real problem. The corresponding IDL operation returns only the value part of the object grouped in a specially created IDL structure. The structure fields can be changed by the client that will not affect the target object. The exceptions are objects, which are returned by a reference (CORBA reference). If the first method (mapping by a reference) is used, then a pointer/reference to a constant variable returned by a C++ function can be converted to a non-constant IDL reference through which the variable can be modified. The problem here emerges for C++ functions returning a pointer/reference to a constant object. How to map such references in IDL?

There could be several solutions, but none of them was good enough to be included in the default mapping. First, a special exception (e.g. XDA_ConstantObject) could be raised whenever an attempt to invoke a non-constant method on a constant object is done. This will make the mediators more complex, since a check operation for the object's constancy will be required in every method that makes modifications. Another possible solution could be returning a reference to a specially defined distinct mediator, which would contain only operations corresponding to constant methods of the related C++ class. However, besides looking queer, this approach requires from the programmer the implementation of additional mediator classes and corresponding efforts for management of their objects.

In conclusion, it was decided to leave the final resolution to the programmer who will choose the proper solution depending on particular application's specific needs.

The Problem with Arrays passed by Pointer

Let us see an example of a C++ function defined in the following way.

```

// C++
void foo(long* tab, int sum);
{
    for(short s = 0; s < sum; s++)
        cout << "the " << s << " element = " << tab[s];
}

```

This is a valid C++ function. As the first argument, it takes an unbounded array, whereas the length of the array is passed as the second argument. This semantic is not obvious from the function's signature. It is implementation-specific knowledge that cannot be expressed via an interface. C++ has also no any additional mechanisms to detect the length of an array passed as a pointer (unless it is an array of chars, which must always be terminated by a NULL character). Therefore, it is not possible to make any assumptions about the length of the

array automatically by default. For example, if the illustrated function would mapped to IDL in this way.

```
// IDL
typedef sequence<long> longs;

void foo(inout longs tab, in long sum);

// C++, implementation
void foo (longs& tab, CORBA::Long sum)
{
    long* l = new long[tab.length()];

    for (short i = 0; i < tab.length(); i++)
        *l++ = tab[i];

    m_ref->foo(l, sum); // error - tab.length() can be > as sum
};
```

It would be impossible on the ORB client side to create an unbounded sequence of long integers and pass it to the server because the length of this sequence is unclear. It is not obvious that the second parameter (long) contains the length of an array passed in the first parameter. On the other hand, using a CORBA's pointer in mediator directly is also dangerous.

```
// IDL
typedef long longArray[10];

void foo(inout longArray tab, in long sum);

// C++, implementation
void foo (longArray_slice * tab, CORBA::Long sum)
{
    m_ref->foo(&tab, sum); // error
};
```

If the second argument is used as described above, then it can result in dereferencing invalid data and can have disastrous consequences.

The postulate was admitted that it is impossible to define default mapping rules for all cases where arrays passed by pointer. In special cases, the programmer's manual intervention is inevitable.

4.3.2.11 Object by Value – struct-based interoperable Representation

Since the server and client can use different programming languages, there is not possible to make general, universal assumptions about representation of objects by value for all environments. Therefore, the standard approach to passing an object by value is to flatten the state data into a struct and pass that through an operation call.

In some cases where an object should be returned by value, a new IDL struct containing all attributes of this object is created and returned instead of the IDL object reference. Thus, only the value part of object is passed, and, in fact, object is copied, not moved (Table 13).

C++	IDL
<pre> Class foo; class factory { public: foo* obj_ptr; foo& obj_ref; foo* newObject (const char * name); foo& getObject (long number); }; </pre>	<pre> struct XDA_fooRep { XDA_foo_obj_ptr; string name; long id; }; interface XDA_factory { attribute XDA_fooRep obj_ptr; readonly attribute XDA_fooRep obj_ref; XDA_fooRep newObject (in string name); XDA_fooRep getObject (in long number); }; </pre>

Table 13. C++ to IDL mapping for pointers passes by value
(struct-based representation)

The Problem with Object References

The IDL struct is a poor substitute for object by value. It carries several limitations and is very different to the interface type. IDL interfaces and structs are not similar constructs as they are in C++. The IDL interface is more akin to a C struct than a C++ struct. It contains only data, cannot have operations and cannot inherit from another struct. So the only object-oriented type in IDL is the interface. However, it can never be passed by value.

As an illustration, consider a client application that must provide a list of objects to a client from which the client can select a specific object for further inspection or modification. The objects are defined as interfaces in IDL.

```

// IDL
typedef sequence <XDA_Object> XDA_ObjectSeq;

interface GTA_Extent : XDA_Object {
    ...
    XDA_ObjectSeq get_objects ();
    ...
};

```

The `GTA_Extent` interface returns a sequence of object references to the client when the `get_objects()` operation is called. However, this will result in the instantiation of proxy objects for every `XDA_Object`. Creating CORBA objects is more expensive than creating non-distributed objects. The performance of the application is composed even more by the need for the client to make remote calls to receive information from the `XDA_Object` objects to populate the list. After all this network activity, the client might make invocations on only a small number of the `XDA_Object` proxies that were created. Working with many objects it is much more efficient to stringify all object references in the sequence and send them in one call. Using the function `string_to_object()` a client can convert them back and instantiate corresponding proxies if (and when) they are needed.

```

// IDL
struct XDA_ObjectRep {
    string nameStr;
    string iorStr;
};
typedef sequence <XDA_ObjectRep> XDA_ObjectRepSeq;

```

```

interface GTA_Extent : XDA_Object {
    ...
    XDA_ObjectRepSeq get_objects ();
    ...
};

```

The single call to `get_objects()` now return enough information to populate the list. However, if the client then wishes to invoke further operations on a particular `XDA_Object`, the client code must first convert the string `iorStr` in IOR reference by calling `string_to_object()` and then narrowing it to the appropriate interface type. Unfortunately, the struct cannot have operations defined on it. If it did, we could provide a more elegant solution.

```

// IDL
struct XDA_ObjectRep {
    string nameStr;
    XDA_Object get_object(); // not legal IDL
};

```

The `get_object()` operation would be invoked by the client to return the `XDA_Object` reference (the proxy). It eliminates the converting of the stringified object reference that would otherwise have to be made. Of course, the IDL above is not legal. The IDL struct is definitely not an object and cannot have member operations.

The Problem with Inheritance

The lack of inheritance in structs is another limitation. Consider an extension to the `XDA_ObjectRep` above for a derived type of `XDA_Object` – `XDA_SpatialObject`. Defining `XDA_SpatialObjectRep`, ideally, we would like to inherit from the `XDA_ObjectRep`. However, since it is not available, other less elegant solutions are used. One approach adds the `XDA_ObjectRep` as an attribute of `XDA_SpatialObjectRep`.

```

// IDL
struct XDA_SpatialObjectRep {
    double x, y, z;

    // attributes of XDA_ObjectRep
    XDA_ObjectRep m_parent;
};

```

Nevertheless, this has inverted the inheritance and turned it into containment. A second solution, expressing the same relation, adds the members of the `XDA_ObjectRep` to the „derived“ struct `XDA_SpatialObjectRep`.

```

// IDL
struct XDA_SpatialObjectRep {
    double x, y, z;

    // attributes of XDA_ObjectRep
    string nameStr;
    string iorStr;
};

```

However, with both of these solutions it is not possible to return the value of an `XDA_Object` instance (base or derived) without adding extra operations to the `XDA_Extent` interface.

```
// IDL
interface GTA_Extent : XDA_Object {
    ...
    XDA_ObjectRepSeq get_objects ();
    XDA_SpatialObjectRepSeq get_spatial_objects ();
    ...
};
```

If we want a single operation to return the values pertinent to the derived types of `XDA_Object`, then we have to modify the `XDA_ObjectRep` struct to carry the derived information. However, since some new specialisations of `XDA_Object` can be created in the future, we have to use the `any` type.

```
// IDL
struct XDA_ObjectRep {
    string nameStr;
    string iorStr;

    // attributes of all derived interfaces
    any m_derived;
};
```

This design enables the `get_objects()` operation to return the value for all derived types of `XDA_Object`. Nevertheless, it is not very elegant and it requires the client programmer to write code to „unmarshalling“ the struct from the `any`.

The struct approach to passing objects by value is somewhat restricted and does not address the object implementation. Transmission of the state data is possible but the implementation of the objects in the sending and receiving process must be compatible. The object in the receiving process must be capable of being constructed and initialised correctly from the data in the struct.

Often the list operation is optimised in the server as a query against persistent storage that returns a result set of objects. These objects may be not CORBA objects but they are definitely objects (not just flat data structs). Therefore, the objects have to be converted to structs before the transmission across the CORBA transport. Often the original result set object structure is reconstructed and the objects are initialised from the struct data. This extra „marshalling“ is expensive both from the development viewpoint (someone who has to write the conversion code) and during runtime invocations. At least this work around is CORBA compliant and interoperable with other middleware systems, for example Java RMI.

4.3.2.12 Object by Value – object-based alternative Representation

Until CORBA 2.1 [OMGorb97] the struct-based representation was the single possible representation for object's values. This allowed networked objects to be perceived purely in terms of IDL structures. However, sometimes the ability to pass an actual copy of an object, rather than a structure is advantageous.

The booming phenomenon of the Java language attests the soundness of the object by value approach. The crucial point of languages such as Java, which enable the object by value paradigm, is that they are interpreted languages as opposed to compiled languages such as C++. Interpreted languages tend to defer almost any decision at runtime thereby increasing the flexibility of the code but at the cost of increasing the runtime overhead. On the other hand, there is no doubt that C++ code is faster than Java code, though Java is more flexible than C++.

Recall that in the standard IDL to C++ mapping, when a reference to an IDL interface is passed as an argument in a remote invocation, a smart local C++ „proxy“ object is created

that encapsulates the logic for interaction with the remote interface. The straightforward idea is to make this local C++ object even smarter: instead of merely encapsulating the logic for the remote invocation, to make from the „proxy“ object itself a precise copy of the remote object so that it can locally execute any operation without making a remote invocation. Note that the client is not aware that the object has been copied. The only thing it deals with is a C++ object pointer regardless of whether this pointer refers to an object that merely represents a proxy to a remote object or is a real server object in its own right.

Following these ideas, OMG has defined the *Object By Value* (OBV) model [OMGorb01] that is based on object copying rather than class migration. In the proposed model, the code is not downloaded at runtime because we are in a compiled environment. Nonetheless, a looser form of object by value approach can be achieved by copying an object instance from one place to another. However, this mechanism implies that the receiving place already has the implementation class of the object to be copied, or else it may be dynamically linked if the system supports DLLs. What is copied is not the implementation class but the internal state of a particular object instance. What are the advantages of this mechanism? Plainly, by copying the object instance, an interaction with that object will be locally resolved instead of causing a remote communication.

The specification adds some new keywords and types to the IDL syntax to allow object state to be defined. The main new definition here is the `value` type. It does not necessarily inherit from CORBA object, can have operation definitions, and explicitly supports the declaration of state data. Here is an example.

```
// IDL
value XDA_ObjectVal {
    string nameStr;
    string iorStr;
};

interface XDA_Object {
    XDA_ObjectVal get_value ();
    void set_value (in XDA_ObjectVal val);
};
```

The above example is similar to the struct shown earlier. The difference is that value types can inherit from other value types and they can have operations. Therefore, the both following examples would be legal.

```
// IDL
value XDA_ObjectVal {
    string nameStr;
    XDA_Object get_object(); // it is now legal
};

value XDA_SpatialObjectVal : XDA_ObjectVal {
    double x, y, z;
    ...
};
```

The proposed value type provides a solution to the limitations of the IDL struct described above when it is used to pass state by value. A public keyword is also proposed for the state members, which are private by default. The value type can „inherit“ from interfaces. Thus, the implementation of an interface type can be defined within the IDL.

A second new type, the abstract interface, is also proposed. An abstract interface is a type that may be passed by value or by reference at runtime.

```
// IDL
abstract interface XDA_Object {
    ...
};
```

The keyword „abstract“ changes the marshalling of the `XDA_Object` type. Existing IDL interface declarations are not affected, they will remain always pass by reference. However, this simple change to the IDL will also enable to use the new by value capability.

Changing existing IDL structs to value types should be straightforward. The real test will be migrating existing object implementations. The specification has tried to accommodate this with interface style TIE and inheritance mappings for the value type that will connect object implementations to the classes generated from the value declarations.

To summarise, when an object reference is passed as an argument in a remote invocation, the state of an object to which this reference refers is copied in the invocation and a new instance is created at destination and initialised with the state received to recreate a precise copy of the object. This mechanism realizes a pass-by-value semantics of objects as opposed to the classical pass by reference, which may be useful in circumstances where performance is a crucial issue. In our model, the pass-by-value semantics does not supersede the pass-by-reference semantics. Rather, both options are available and the client cannot tell which has been applied since the mechanism is completely transparent: all the client deals with is a C++ pointer that may either correspond to a proxy object or be a precise copy of the remote object. This transparent behaviour permits greater flexibility because the client is not affected in both cases. Only the copied object is indeed aware of the mechanism and must cooperate to achieve the object by value semantics.

The Problem with Interoperability

Although the Object By Value model increases the flexibility afforded to distributed system designers, value objects are no longer perceived in terms of a public interface. Their interface is compounded with the definition of the state: the CORBA IDL has been extended to include the value type that defines this state.

Further, the new model reduces the interoperability of the CORBA model. Value type implementations are always local. That is, they are not CORBA objects that are registered with an ORB, but local programming language entities (for example an instance of a C++ class). However, if a value type supports an interface type, as long as it has been registered with the ORB it can also support CORBA's usual object by reference semantics.

When an object is passed by value, its state is marshalled, the receiving context creates an instance of the object and the state is unmarshalled into this new instance. Three possible situations can occur when a value type is passed from a sending context to a receiving context.

1. The receiving context could support the implementation directly.
2. The receiving context does not support the implementation directly, but can upload an appropriate implementation (possible from the sending context).
3. The receiving context does not support and has no way of obtaining the implementation.

These three scenarios are depicted in the following Figure 27. The programming language used in the implementation will affect the actual mechanism employed. For example, most languages can support the implementation directly through the implementation of additional value classes in the receiving context (scenario 1). Scenario 2 can be used in languages with

intrinsic support for code mobility, for example Java. If neither value classes nor mobility can be supported, scenario 3 will be the result.

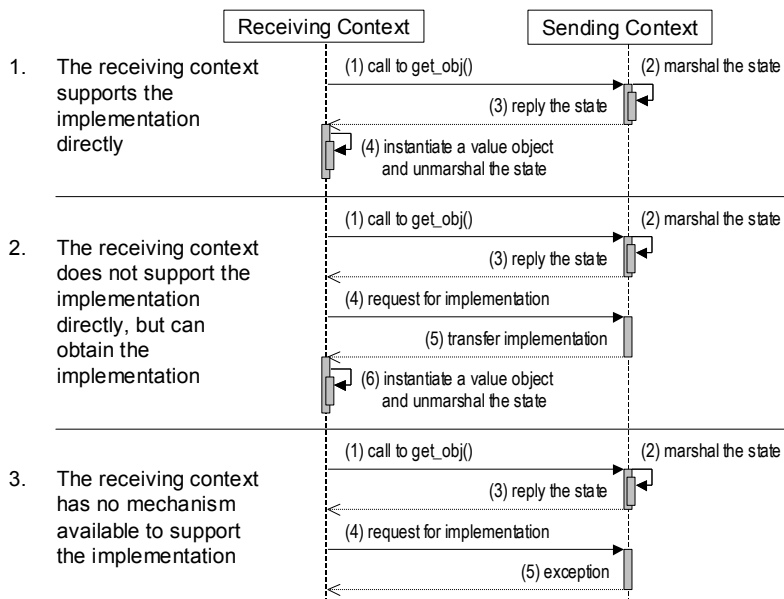


Figure 27. Three Scenarios for Object By Value

The Problem with Semantics of Attributes

The *Objects By Value* specification uses type structure equivalence to determine whether two object implementations are equal (more specifically, if the receiving context can support the implementation of the object directly). The structure of the IDL source is parsed to produce a 64-bit signature (using a hashing algorithm operating over the types within the source). If the receiving context expects a different signature from that received, it can deduce that its definition of the object has a different type structure to that of the sending context. In this case, it can either fail the operation or attempt to fulfil the request using information from the interface repository. If the signature is the same for sending and receiving contexts, it is highly probably that the two objects are type-structure equivalent. However, there is a potential problem with this definition of equivalence. Consider the following two IDL code segments.

```
// IDL
interface Weight {
    int killos;
    int grams;
};

interface Weight {
    int pounds;
    int ounces;
};
```

Although both these code segments would result in the same signature being generated by the hashing algorithm (and therefore be taken as „identical“ by implementations), it can be seen that they are clearly different.

4.3.2.13 The Problem with collection-related Interfaces

A mediator representing a collection operates properly only when the collection is either (a) a transient collection with elements pointing to transient memory or (b) a persistent collection with elements pointing to persistent memory. Other types of collections cannot be supported by the default implementation. The lifetime of mediators for collections is managed by the adapter in the standard way. Indeed, if elements of a transient collection are pointers to persistent data, they are discarded as soon as the server makes shutdown. Since it is impossible to determine the kind (persistent or transient) of a collection, for example,

returned by a database method on the basis of method’s signature, the adapter does not make any assumptions about the persistence of manipulated objects leaving possible corrections to the programmer. If, for instance, a transiently allocated collection of pointers to persistent objects is returned by a database method, then the best idea is to transfer these pointers as CORBA references to the client.

Additionally, if a pointer/reference to a constant collection should be returned to the client, then the above-mentioned problem arises for handling references to constant objects. Since the operations, which can potentially modify the collection object (such as insert element, delete element), as well as their default implementation is provided by the adapter it was decided to adopt the approach introducing a new exception (`XDA_ConstantObject`). That exception is raised whenever an attempt to modify constant collection is made.

4.3.3 Execution Model

This section defines the concepts associated with implementation of the defined abstract model i.e., the concepts relevant to realising the behaviour of abstract CORBA objects in a computational system according to the developed rules for mapping of persistent C++ into CORBA’s IDL.

4.3.3.1 Interfaces, Types and Classes

The interface of the original database object is available to CORBA clients through the IDL interfaces of additional transient CORBA objects – mediators. The basic question here is: how are these transient CORBA objects related or bound with corresponding persistent database objects?

Let us define some terms that we will use in the following discussion. According to the defined persistent object model, a *class* has one or more *IDL interfaces* and null or one *implementation*. The interface of a class is specified in the Interface Definition Language (IDL) according to the rules defined in the previous chapter. Where a class defines an association between several interfaces and one implementation, a *type* is only a part to the class, defining a particular one to one association between an interface and the implementation (Figure 28).

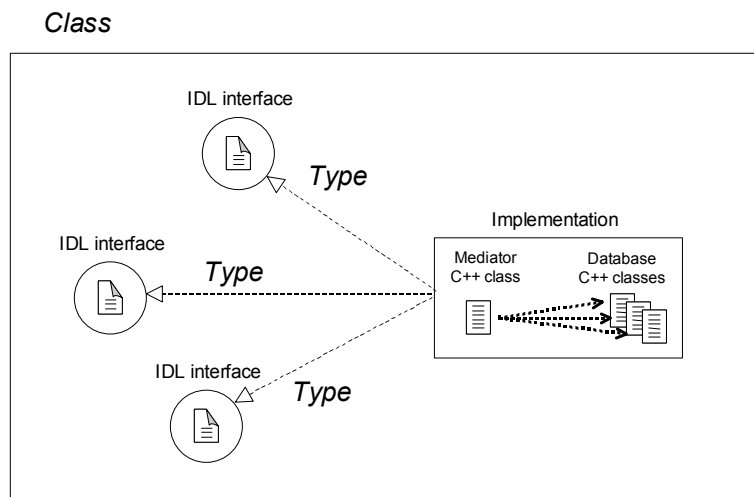


Figure 28. Type and Classes

Since a CORBA client sees only IDL interfaces, it manipulates with persistent objects using types. Therefore, the *type* in our object model has the similar semantic as the *IDL type* in CORBA. The *class* is mainly used in the server side and defined through the mediator’s C++

class, which plays the central role in our model binding IDL interfaces together with corresponding implementations.

The implementation part encompasses the data members and member functions actually defined by the IDL interface. The implementation has two parts: a mediator C++ class and a set of associated database C++ classes. In this schema, the mediator class implements the IDL interface and acts as a usual CORBA servant class. It consists of IDL interface specific member functions and data members implemented with the help of functions provided by the associated database classes. Therefore, usually mediator classes have no state, besides references to instances of the database classes.

Interface Granularity

Mediators are responsible for making the database heterogeneity transparent to CORBA applications. Retrieving and combining data from underlying data sources the mediators give applications an integrated view of data, and thereby decoupling them from the necessity to understand multiple data models. The question is how the methods accessible on database classes can be grouped into CORBA IDL interfaces. This aspect is usually referred in the literature using the term *interface granularity* [Ami98, HV99].

The interface granularity is a topic of predominant importance for the integration of database management systems with CORBA environments since it has a great impact on the scalability of the integrated system. Consider a database schema containing a large number of small persistent classes. Applying the default 1:1:1 binding to these database classes we will receive the same large number of IDL interfaces and, as result, mediator classes (see case 1 in Figure 29). The number of available mediator classes implicitly determines the number of active mediator objects, which in it's turn is limited by the amount of the available memory. So, in the case of many database classes (and objects) the default 1:1:1 binding between databases classes, mediator classes and IDL interfaces can be extremely inefficient and negatively influence performance and scalability aspects of the integrated system.

The solution to this problem is to reduce the number of mediator classes. When the number of mediator classes is independent of the number of IDL interfaces and database classes, it is possible to reduce the number of mediator classes combining relevant features of several database classes into one single mediator class and presenting them to the client through one or several IDL interfaces. This type of the binding is named as n:1:n binding (n IDL interfaces : 1 mediator class : n database classes) or *views* (see case 2 in Figure 29).

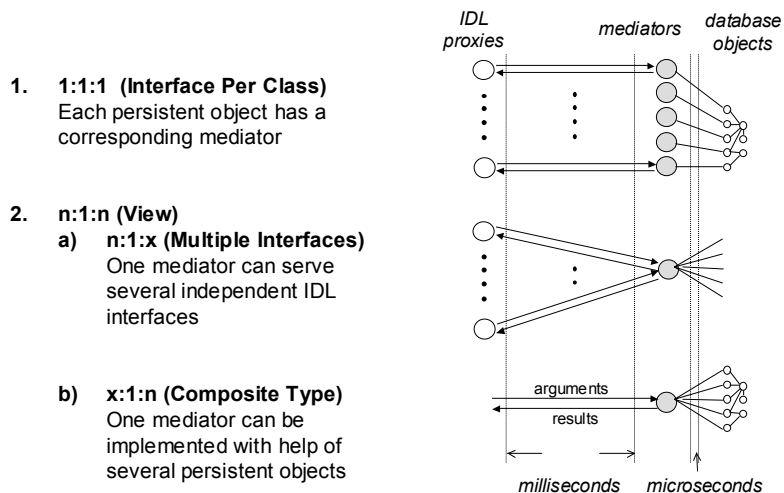


Figure 29. Binding of mediators to database objects

In general, the primary motivation for defining views is to include sharable semantics and structure in the information. When a database is shared among multiple applications, these applications typically have differing requirements for data access and representation. Such differences are supported by having views, which present diverse subsets of the base data. Must all applications sharing objects use the same object schema, or is it better to give each application its own object schema and somehow to integrate them?

If multiple applications differ in view, the needed compromise reduces the relevance and effectiveness of the object representation. For example, note a single viewer application that manipulates information residing in different repositories and a single repository that may serve different viewer applications. If the storage aspect (database classes) is separated from the presentation aspect (interfaces), then it is possible to reduce the number of mediator classes by factoring out the behaviour along the two dimensions. The worst-case scenario one needs to create $m \cdot n$ mediator classes (where m is the number of database classes and n is the number of viewer applications) while one can do with k : $k < m$, $k < n$ mediator classes using views.

Defining views, typically, the programmer chooses a subset of large-grained database classes, which interfaces will be exposed to the CORBA clients and also represent the other database objects (fine-grained ones). However, implementing mediator classes, the programmer is not limited by the existing database classes and is free to define a new, completely different IDL „image“ of the database. For example, using views it is possible to define IDL interfaces providing several different representations of the same database classes (multiple interfaces, case 2a in Figure 29) or defining completely new interfaces through compositions of several database classes (composite types, case 2b in Figure 29). In sum, XDA model defines two basic types of binding between database classes and IDL interfaces.

- I1: 1:1:1, Interface Per Class.** One CORBA interface per database class is defined, including all access method and type definitions. This is not reasonable in the case of R1 (as R1 does not know objects), well possible for R3, where the structural part of each class is defined in one struct, and necessary for R2, where database attributes are mapped to interface attributes.
- I2: n:1:n, View.** Relevant features of several database classes are combined in one single mediator class and presented to the client through one or several IDL interfaces. Here we are identifying two sub cases.
 - a) n:1:x, Multiple Interfaces.** Relevant features of one or several database classes are presented to the client through several different IDL interfaces. This is reasonable for systems using R2 or R3; to different CORBA clients the same database object can provide different „views“ of his state. R2 requires the definition of separate interfaces for each class.
 - b) x:1:n, Composite Types.** Relevant features of several database classes are combined in one single interface. This is reasonable for systems using R1. The extreme case is an interface having only one method that takes queries as strings, which are passed to the database server. The results of queries are represented as strings also. Using R3, the struct will probably get uncomfortably large, as it must include the struct definitions of all classes. R2 requires the definition of separate interfaces for each class.

4.3.3.2 Objects

In our object model, *objects* are the instances of the *classes* defined according to the schema presented in the previous section (Figure 28). Practically, an instance of the class is a couple including an instance of the mediator class and a set of instances of associated database

classes. Even though „persistent CORBA objects“ are not fully kept in persistent memory, to their CORBA clients they appear as long-lived objects, until the application server takes care about transient mediator objects, their dynamical instantiation and releasing. Since the mediator objects serve only as a „glue“ between ODBMS and CORBA environment, they usually have no state besides references to the associated persistent database objects. Therefore, the instantiation of a mediator object does not lead to the instantiation of the whole object. They adapt the format of C++ interfaces of persistent objects to CORBA-compatible format required by the interfaces of skeleton objects, which are generated by the IDL compiler. For the sake of brevity, those formats can be marked as ODBMS and CORBA formats respectfully.

The primary responsibilities of mediator objects are the following.

- fitting the signatures of operations (arguments, return values) of pre-existing persistent objects and generated skeleton objects;
- creation of data structures necessary for carrying parameters/return values of operations;
- creation of a CORBA IOR when a reference to persistent object must be returned to the client;
- assignment between ODBMS and CORBA variables;
- persistent object's method invocation;
- transforming persistent ODBMS complex types (e.g. `os_Collection`) to appropriate CORBA complex types' (e.g. `sequence`) C++ mappings;

More complex operations like queries, iterations and data conversions can be supported by mediator objects also.

The basic Functionality of Mediator Objects

The *basic* functionality of mediator objects can be roughly divided onto three phases.

1. pre-invoking phase: processing the input parameters of the current request, i.e. „translating“ input parameters from CORBA format into ODBMS format; in particular, `in` and `inout` arguments of methods as well as the new values of CORBA attributes are processed;
 - data values are converted from CORBA format into ODBMS format, creating the necessary transient ODBMS objects and assigning to them the values of CORBA input parameters;
 - input CORBA IOR references are translated to equivalent ODBMS references;
2. invoking the methods or updating the attributes of persistent objects;
3. post-invoking phase: processing results of the current request, i.e. „translating“ the results from the ODBMS format into CORBA format; in particular, return value, `out` and `inout` arguments, as well as the existing values of ODBMS attribute are processed;
 - data values are converted from ODBMS format into CORBA format, creating the necessary transient CORBA objects and assigning to them the values of ODBMS output parameters;
 - output ODBMS references are translated to equivalent CORBA IOR references.
4. returning method invocation result or attribute value to the client;

It should be noted, that only step 2 is performed always. Other steps are optional. Step 1 is required when (A) there are `in` or `inout` parameters and (B) there is no default conversion from CORBA to ODBMS types (e.g. primitive types have default conversion, for more details please see Section 4.3.2). Similarly, step 3 is required when (A) there are `out` or `inout` parameters and (B) there is no default conversion from ODBMS to CORBA types. Step 4 is performed always when an operation returns a value. The step 2 is performed only if an operation does not take any parameter and does not return any value (e.g. `void dummy()`) or if default conversion exists for all its parameters/return value.

Additional Actions taken by Mediators

Apart from the basic functionality, some additional operations can be performed by mediator objects in order to achieve various tasks, such as reduction of remote network calls, clients' authorisation or statistics measurements. Some of these operations should be supported by the application server; others can be added to the mediators' code manually by programmers.

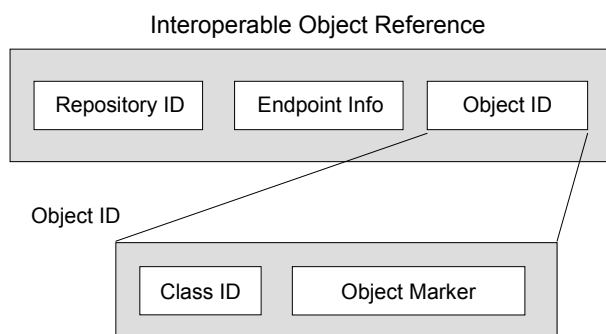


Figure 30. Interoperable Persistent Object Reference (IPOR)

4.3.3.3 Objects Identity

Object Identifiers (Object IDs) are related to objects and establish their identity. CORBA clients work with remote server objects through *Interoperable Object References* (IORs) that identify remote objects in the CORBA environment. Since persistent objects are different from usual CORBA object having a rather complex structure (Figure 29), usual IORs are not enough for their identification. Preserving the compatibility with the OMG object model we have implemented *Object IDs* by extending CORBA's IORs and name them as *Interoperable Persistent Object References* (IPORs).

Such references are fully conformant to the current OMG model and differ only by the *Object ID*, which has a propriety format (Figure 30). *Object ID* contains two basic components: *Class ID* and an *Object Marker*. The *Class ID* defines the class of the object keeping the name of the corresponding C++ mediator class, where the *Object Marker* identifies the associated database objects keeping for each of them the name of its database and its reference within the database (a stringified reference of the `os_Reference` class' family).

Operations on Interoperable Persistent Object References (IPOR)

The information provided by IPORs is sufficient for the unique identification of distributed persistent objects within the CORBA environment. Actually, the uniqueness of the IPORs directly follows from the uniqueness of the IORs within the CORBA environment, C++ class names and database references. So if the application server receives such an IPOR reference, it can obtain from the reference all information necessary for the location of the associated database objects and (construction, if deactivated) a transient mediator.

Compatibility of IPORs with usual CORBA IORs allows to use on them all operations defined for normal IORs. Some basic of them are the following.

Nil object references. Nil references are analogous to C++ null pointers and contain a special value to indicate that the reference points nowhere. Therefore, it is not possible to invoke operations on a nil reference. Only `CORBA::is_nil()` function can be used to test whether a reference is nil. All other attempts to test for nil have undefined behaviour.

Determine the interface (`_is_a()` operation). The `_is_a()` function is similar to `_narrow()` in that it lets to determine whether an object supports a specific interface, for example.

```
CORBA::Object_ptr obj = ...; // Get a reference
if (!CORBA::is_nil(obj) && obj->_is_a("IDL:Bank/Account:1.0"))
    // It's an Account object...
else
    // Some other type of object...
```

The test for nil in this code example prevents the client program from making a call via a nil reference. `_is_a()` lets applications manipulate IDL interfaces without static knowledge of the IDL – that is, without having linked the IDL-generated stubs. However, most applications have static knowledge of IDL definitions, so they can use `_narrow()` and never need to call `_is_a()`.

Narrowing object references (`_narrow()` operation). The process of converting an object reference for a base interface to a reference for a derived interface is known as narrowing an object reference. To narrow an object reference to class `T`, a `T::_narrow()` operation must be used. If the parameter passed to `T::_narrow()` is not of class `T` or one of its derived classes, the operation returns a nil object reference. For example, to narrow an object reference `obj` from the previous example to `Account` reference.

```
Account_ptr aPtr;
if (aPtr = Account::_narrow(obj))
    // It's an Account object...
else
    // Some other type of object...
```

Widening object references. The process of converting an object reference for a derived interface to a reference for a base interface is known as widening an object reference. In most programming languages, widening is made implicitly supporting normal inheritance conversions.

Check existence of objects (`_non_existent()` operation). The `_non_existent()` function tests whether a CORBA object exists. It returns true if an object no longer exists. Invoking an operation on a reference to a non-existent object is safe, but the client must be prepared to handle exceptions.

Check identity of objects (`_is_equivalent()` operation). The `_is_equivalent()` function tests whether two references are identical, i.e. that both references point to the same object. It should be noted, that with our approach to defining an IPOR, we can distinguish between object identity and object equality. Two objects are equal if their states are the same (denoted by „`==`“). Two objects are identical if and only if they are the same objects (denoted by „`==`“): that is, their IPORs are the same. Because `_is_equivalent()` tests for absolute identity of object references, using this operation for IPORs it is possible to check only identity of persistent objects.

String conversions (`object_to_string()` and `string_to_object()` operations). Object references can be converted to and from strings, which facilitates persistent storage. When a client obtains a stringified reference, it can convert the string back

into an active reference and contact the referenced object. The reference remains valid as long as the object remains viable. When the object is destroyed, the reference becomes permanently invalid.

Advantages of IPORs

There are several advantages to using such an IPORs as the mechanism for object identity.

- *They are efficient.* IPORs require minimal storage within a complex object. Typically, they are smaller than textual names, foreign keys, or other semantic-based references.
- *They are fast.* IPORs point to an actual address or to a location, which gives the address of the referenced object within the database. This means that objects can be located quickly whether they are currently stored in local memory or on disk.
- *They cannot be modified by the user.* If the IPORs are system generated and kept invisible, or at least read-only, the system can ensure entity and referential integrity more easily. Further, this avoids the user having to maintain integrity.
- *They are independent of content.* IPORs do not depend upon the data contained in the object in any way. This allows the value of every attribute of an object to change, but for the object to remain the same object with the same IPOR.

Note the potential for ambiguity that can arise from this last property: two objects can appear to be the same to the user (all attribute values are the same), yet have different IPORs and so be different objects.

- *They are persistent.* The architecture of the application server is quite powerful and additionally allows, besides providing persistence to CORBA objects, to provide persistence to the corresponding object references. This means that a client that has an object reference can use it at any time without warning, even if the mediator has been deactivated or the server system has been restarted. Driven by incoming requests, the application server activates objects that lie dormant in persistent memory.

For example, with the help of string conversion operations a client can save a persistent IPOR in a file, make shutdown and then in another run load this reference from the file and use it again.

```
ofstream fout("reference.ref" );
fout << orb->object_to_string(objVar) << endl;
// ...
objVar = orb->string_to_object("relfile:/reference.ref");
```

With the persistence of object references, it makes perfect sense for the client to store an object reference for later use or send it to other client. For example, references to the persistent CORBA objects implemented by a server X can be stored by a server Y (a client of server X) and vice versa, thereby enabling the construction of ORB-connected multidatabases.

4.3.3.4 Representations

We distinguish three different ways in which database objects can be represented in the CORBA world. First, database objects are treated as binary data streams and are not interpreted in any way by the CORBA middle layer. Second, the data of a database object is stored in a struct, which conforms to the database class definition. Third, each database object is represented by a corresponding CORBA object reference.

- **R1: As Raw Data.** In this case, database objects do not exist as identifiable entities in the CORBA world. Concerning the ORB, they are unstructured binary data streams.

These streams must be parsed by the client using a fixed encoding of their inner structure.

R2: As Object References. In this approach, each database object is represented by an equivalent CORBA object reference. Therefore, database references are translated directly into CORBA references, which can be carried out automatically. If a client requests a database object, he will get back an identifier, but not the actual data, which remains on the server. Each attribute access will result in a remote call, which increases network traffic but avoids certain problems of outdated and large-scaled data.

R3: As Structs. The structure of a database object can be represented in an IDL struct by simply copying all attributes. IDL specifically supports sequences which can model multivalued attributes. Upon requesting a database object, the client gets back a struct equivalent to the class definition. Structs can be nested to represent complex attribute types and inheritance; multi-valued attributes are stored in proper sequences or generic set-classes. IDL structs for each database class can be created automatically using the schema information. Structs carry the object's data to the client preserving the type. All subsequent operations are performed locally, reducing network traffic and enhancing performance, but complicating integrity preservation in case of multiple clients working with the same object.

R4: As Objects. In the proposed model, a looser form of object by value approach can be achieved by copying an object instance from one place to another. However, this mechanism implies that the receiving place already has the class implementing the object to be copied, or else it may be dynamically linked if the system supports DLLs. What are the advantages of this mechanism? Plainly, by copying the object instance, an interaction with that object will be locally resolved instead of causing a remote communication.

4.3.3.5 Collections

A collection is an object whose purpose is to group together other objects. It provides a convenient means of storing and manipulating groups of objects by supporting operations for inserting, removing, and retrieving elements. Collections are implemented as usual persistent objects (see Section 4.3.3.2 above) with the help of a library of collection classes provided by ObjectStore (see Section 2.1.2.3). These classes provide the data structures for representing such collections, encapsulated by member functions that support various forms of collection manipulation, such as element insertion and removal. Retrieval of particular collection's elements for examination or processing is supported using a cursor class.

Many application types require two forms of data access: navigational access and associative access. Navigation accesses data by following pointers contained in data structure fields. In C++, the syntax of access to data members supports navigational data access. Associative access, on the other hand, is the look-up of those data structures whose field values satisfy a certain condition (for example, look-up of an object by name or ID number). Collections support associative access, or query, through member functions in the ObjectStore class library.

4.3.3.6 Queries

Queries are implemented on collections, which at the same time serve to group together those elements of a given collection that satisfy a specified condition. The query processing database service provides support for associative data retrieval, such as look-up by name or

ID number. Although it is often desirable to export all these possibilities to clients, there are many applications where this is not necessary.

We distinguish three basic approaches: queries are either hard coded inside the mediators, or they are restricted to result in proper database objects, or clients can pose arbitrary ad-hoc queries.

Q1:Fixed Queries. Fixed queries are a common method to hide all implementation aspects of the database to the client. The client does not need to have any knowledge of the database schema, the query language, the access path etc., but just has the choice between a certain number of methods, each representing a fixed, though possibly parameterised query. In general, the results of these methods are not restricted by the database schema, as arbitrary processing can be carried out in the embracing method's code.

Q2:Class-restricted Queries. This approach captures all queries of the form: „Give me all objects of class X which have ...“. This has a number of consequences.

- The result is always an uniquely identifiable database object. Restructuring queries are not possible. However, joins are still allowed for the specification of conditions.
- In many cases, projections are vital for better performance, for instance projecting out the image of an employee when the name suffices. Projections are still possible here, but clients must be able to determine whether a value is missing in the database, or whether it was only projected out by the query. The object identifier must always be retained to allow propagation of changes.

This approach is identical to the next type of queries, with the restriction that the „select“ part of each query is bound to attributes of one class only and must contain the object ID. Although this type of queries seems to be quite restrictive, it is certainly sufficient for many applications, for instance digital libraries.

Q3:Arbitrary Ad-hoc Queries. Here, clients are able to pose arbitrary queries, exploiting the full power of the underlying database query language. Queries are passed as strings, which are directly transferred to the database server for execution.

This type of queries is reasonable in combination with the first type of objects representation, since other types would be extremely expensive, as the middleware would need to parse the query and dynamically determine the type of the result.

Results, which are always objects of a predetermined class, can be represented using one of the specified three types of the representation. Here the R1 makes little sense, as it misses the notion of classes. However, usually the result of a query is a set of entities. Any of the above methods will result in (homogeneous) collections of structs or object references for R2 or R3, respectively. Using the representation R1, the result can be either a collection of strings, or their concatenation.

4.3.3.7 Metaobject Protocol Interface and Metadata

Schema information for ObjectStore databases and applications is stored in the form of objects that represent C⁺⁺ types. These objects are actually instances of ObjectStore metatypes, so-called because they are types whose instances represent types.

The application server provides for CORBA client a *metaobject protocol* (MOP) interface allowing to use the operations provided by the MOP interface of the target ODBMS. The interface is implemented as a set of IDL interfaces that allows the programmer to access ObjectStore schema information in runtime. The application server provides meta-

information to the CORBA client in the object-oriented form of classes and objects, so a user can query this information as he or she can query other objects. Through the mediator types provided by application server the client can access corresponding ObjectStore metaobjects.

The communication between the client and server is performed by the static communication mechanism (SII for the client and SSI for the server). Using this interface a client can access existing persistent objects and ObjectStore database schema at runtime in a unified generic way. Through provided by the application server mediators it is possible not only to read schema information and attributes in runtime, but also to create classes dynamically and add them to ObjectStore database schemas. *Dynamic CORBA interfaces* (DII for the client and DSI for the server) are used for communication with such dynamically created classes or unknown at compile time types.

4.3.3.8 Transactions

All access, creation, modification, and deletion of persistent objects must be performed within a transaction. Leaving to persistent database objects the responsibility of starting and committing (or aborting) transactions is not an option, because accesses to persistent memory happen both before and after these objects' methods are called. In order to delegate an operation to its database object, a mediator must use references to persistent memory, which can be accessed only within a transaction. Moreover, marshalling of operation results into a reply message may involve accesses to persistent memory also.

One possible solution is to use CORBA's *Object Transaction Service* (OTS) [OMGots01]. The usage of OTS would ensure that not just the user provided implementation code, but also the request dispatching and parameter marshalling code are executed within transactions. Since OTS interacts directly with the local resource manager (the ODBMS), transactions would be started and committed (or aborted) externally to the CORBA server. If OTS is absent, the application server must take the responsibility of starting and committing (or aborting) local transactions. We did not have OTS, so this was our scenario.

Concurrent Multiclient Access

The XDA provides a distributed transaction management similar to the OTS. The concurrency control scheme employed for XDA transactions is based on a conventional serialisable transaction management approach supported by ObjectStore. Transactions can be either local (only one server participates in transaction) or global. Like most database management systems, ObjectStore tries to interleave the transactions of different client processes to maximise concurrent usage of resources. The scheduling of transactions performed by ObjectStore conforms to a *strict two-phase locking* discipline (except in the case of *Multiversion Concurrency Control*; see Section 6.4.2). This discipline has been proven correct in that it guarantees serialisability; that is, it guarantees that the results of the scheduling are the same as the results of some noninterleaved schedule of the transactions' operations.

Since the ObjectStore server can control transactions only for different clients, which are separate UNIX processes, the serialisability of transactions from multiple clients is guaranteed only if a particular application server serve only one CORBA client. The application server is implemented as a CORBA server, which association with clients is controlled by the CORBA server activation policy. Therefore, the server activation policy has direct impact on the serialisability of transactions and consistency of concurrent multiclient access.

CORBA Server Activation Mode

CORBA distinguishes three kinds of server activation modes. In case of a *shared* server, client requests for objects having the same location are all routed to the same server process. With

unshared servers, the requests are routed to different server processes. Using the *per-method* mode, each method call results in a new server process. For the first two policies, one furthermore has to choose between *per-client*, *per-client process*, or *multiple-clients* activation, which determines whether calls from different clients or principals are routed to the same or to different servers.

Choosing the server activation policy has an impact on the degree of parallelism that can be achieved for client queries. To increase response time, all major database management systems are already capable of serving multiple connections: clients can send queries in parallel without having to wait for the termination of earlier queries. This property should be propagated to clients connecting through the application server as well. However, the application server, which is a process on its own, can easily become a bottleneck. If the *shared server* mode together with *multiple-clients* is used, all requests will be directed to the same application server. To support transactions serialisability, receiving a request, this server will block all further requests until termination of the first. This should be avoided.

On the other hand, choosing *unshared-mode*, will result in a great number of processes: one process for each access to each database object. The activation mode policies are somehow subsumed by the use of threads in the application server. Therefore, a new version of ObjectStore DBMS supports threads and with the help of sessions it can serialise transactions from different clients within one process. However, programming with threads is considerable more complex than programming with processes. The portability of the threaded applications is affected by the used thread-safe system libraries, which are not available on all platforms. Furthermore, the introduction of sessions is critical only for performance issues bringing no significant changes to the semantic of the concurrent access management. We did not want to tackle with the complexity of thread-safe programming, so *unshared-mode* was taken in our application.

4.3.3.9 Databases

The persistent object model supports the concept of databases as storage areas for persistent objects of a given set of types. A database has a schema that is a set of type definitions. Each database is an instance of type `XDA_Database` with the built-in operations `open()`, `close()` and other operations. Entry points to the database are available through roots, which are the names (roots) bound to an object using the `create_root()` operation, and unbound using the `destroy_root()` operation.

4.4 eXtensible Database Adapter Framework

The most successful system designs described in the previous chapter have in common that ODBMS dependent functionality is hidden behind a portability veneer, and that database-specific adaptor modules are employed to map the veneer onto the interfaces of the underlying ODBMS. In the Three Tier Architecture presented above the application server is the key component. It is the glue between the CORBA server-side API and the DBMS client API, and as such is only present in the middle-tier. However, until now no technical details concerning how an application server could be implemented have been given.

In this section we present the architecture of the eXtensible Database Adapter Framework, which makes possible the implementation of portable and yet efficient application servers. The suggested system design is called the adaptor model [Maff94]. In the context of this thesis, the adaptor model serves to structure the runtime system of an application server.

4.4.1 Architecture of the XDA Application Server

The evaluation of existing methods for the implementation of CORBA/ODBMS integration made in Section 3.6 showed the advantages of the method based on the *Object Database Adapter* (ODA). Therefore, using our own experience with the OOSA/ODAF as an example, we developed an application server development framework – an *eXtensible Database Adapter* (XDA).

The XDA is the central point of the application server architecture presented in Figure 31. In our study we have tried to investigate the most critical issues of the CORBA/ODBMS integration that we missed in commercial adapters (see Section 3.6). In particular, we have tried to develop a framework that can substantially simplify the development of application servers providing ready-to-use blocks and extensible components.

Using the XDA framework for building an application server a programmer implements database-specific mediators and services by reusing or adapting components provided by the XDA (Figure 31). The XDA self incorporates the functionality of the mediating tier by tackling with above-mentioned basic issues for the management of persistent objects to secure the proper cooperation with the ODBMS. The framework is also an attempt to catch the golden mean between automation and hand-coding. The place where the advantages of both approaches are maximally preserved determines the value and applicability of the adapter.

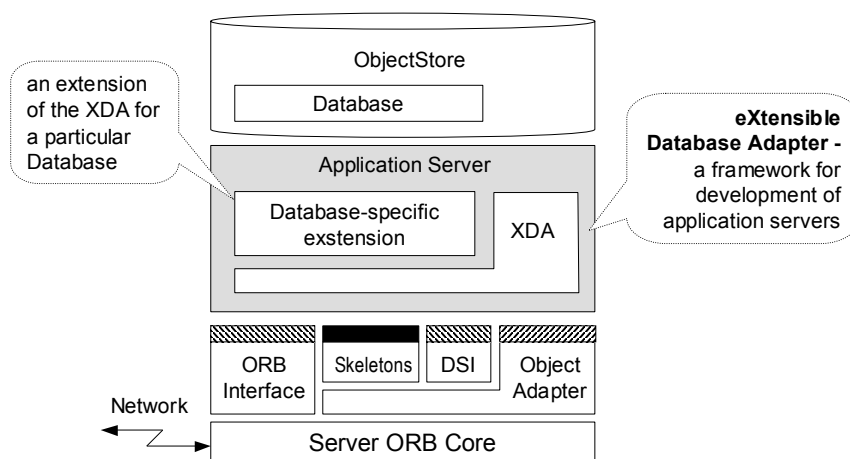


Figure 31. The functional architecture of the application server

4.4.2 The XDA Framework Concepts

Software developers are being asked to build applications in less time and with less money. Object-oriented techniques and component software environments are accepted and in wide use now. Therefore, the generation of software components has attracted more and more software developers all over the world [SUNjb02]. In addition, the use of plug-and-play components from the computer hardware industry has proven the usefulness of predefined interfaces and corresponding implementing components.

Therefore, the XDA architecture is based around a framework model that espouses encapsulation, interfaces and explicit binding in the form of components. The component framework specifies not only the design of a component, but also the way in which components interoperate. There are some reasons why a framework approach for component design is superior to an ad-hoc approach.

- *Frameworks provide guidelines for component designers*; systems that do not mandate a framework for the design of their components are often difficult to work with.

Designers must rely on sample code and may be left unsure as to how implementations may deviate from these examples. Conversely, a framework provides a specification for component design that can be interpreted freely so long as the semantics are obeyed.

- *Frameworks allow a system to control dynamic component behaviour*, by requiring that components divulge information to the system in well-known ways (e.g. by implementing abstract interfaces), it is possible for the system to arbitrarily access this information and use it to control dynamic operations such as component creation and binding.
- *Frameworks specify a consistent pattern for component interoperability*; without a framework, components would interoperate in ad-hoc ways rendering those designed by one organisation incompatible with those from another.

Rather than deploy yet another, the XDA uses a suitably adapted version of the ODAF framework. In addition, influences from other component frameworks that already exist within the literature, e.g. Java Beans [SUNjb02], COM [COM95] and CORBA components [OMGcm02] are also taken into account. At its most fundamental level, the XDA is a simple but highly efficient means of interconnecting language independent components and lends itself well to performance intensive operating system domain. In addition, using an adapted version of a well-known component framework immediately opens up the development of third party components to a large pool of developers. Furthermore, it provides credibility for the project outside of the research community.

4.4.2.1 Components-based Architecture

In XDA, *components* are encapsulated entities, which communicate with other components in their environment through strongly typed interfaces defined in a C++ programming language. Components are used to encapsulate both system and application level activity in a uniform manner. This promotes a consistent development framework that assists application server developers in designing their own system related components if a suitable one does not exist in the current pool. In addition, two particular component specialisations are required to fulfil specific roles within XDA.

- *Mediators*, discussed above, provide communication services between the clients and persistent database objects.
- *Managers* provide services for management of mediators and database objects.

The core of the adapter consists of the main container module *XDA_Adapter* and four replaceable plug-in components – managers (Figure 32).

The adapter represents a public facade for all managers, controls them and allows their replacement. The managers are responsible for the basic relatively independent parts of adapter's functionality. The XDA has four *managers*: an *object manager*, a *database manager*, a *type manager* and a *transaction manager*.

The *object manager* is responsible for the management of mediators for database objects that are defined as user-defined objects, which can be stored in a database. Driven by incoming requests, the object manager should activate objects that lie dormant in persistent memory. To allow on-demand activation of dormant objects, it must ensure that object references handed out to CORBA clients contain information on the location of the corresponding objects in persistent memory. Hence, the managers are usually also responsible for the generation and interpretation of references to objects they manage.

The *database manager* has the functionality similar to the *object manager*, but the database manager is responsible for management of mediators for objects representing databases and internal database structures such as segments and clusters. The manager keeps track of all mediators and controls the segmentation of persistent objects in the database.

The *type manager* is a manager that is responsible for the management of special mediators representing meta-information about available database classes. Such CORBA objects are named as „type objects“. They provide meta-information about the whole set of available mediator classes and their relations to corresponding persistent counterparts (views).

The *transaction manager* is responsible for handling all database updates as atomic and recoverable actions. The manager controls the boundaries of transactions and manages mediator objects that represent database transactions.

For every manager the XDA provides a default implementation. It can be used in most cases. However, if the target database has some specific requirements, for example to the creation of the databases, then the default manager should be replaced by the custom implementation. For this purpose, all managers are designed as plug-in components that can be replaced or extended by the programmer. Since the managers defined above represent relatively independent parts of the XDA's functionality, only a particular manager that is responsible for a given functionality need to be replaced. The XDA requires only that the custom managers should be compatible to the standard interfaces defined by the XDA.

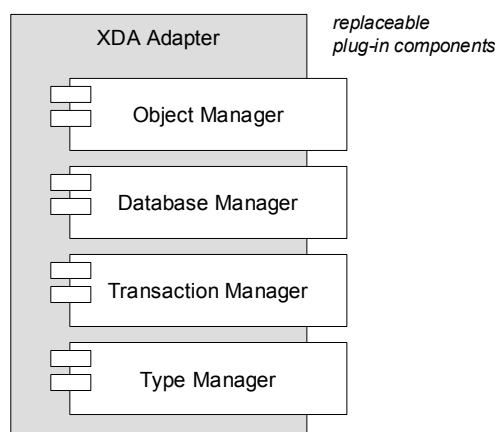


Figure 32. XDA's component-based architecture

4.4.2.2 Implementing Mediator Components

The XDA architecture uses the mediator approach [Wied92, Wied97] assuming heterogeneity of the connected data sources. Data could be stored in different database management systems, using different data models. Moreover, even if the same database is used, data could still be semantically heterogeneous [She91]. The principal issues for the implementation of mediators are (A) the issues of C++ to IDL mapping (analysed in Section 4.3.2) and (B) issues of reducing the costs for their development. Therefore, the XDA framework tries to simplify the implementation of mediators for the programmer providing ready-to-use templates, macros and base classes.

The XDA object manager has the responsibility for managing the instances of custom mediators for database objects. The manager dispatches the incoming requests locating the instances of compatible mediators that can process the requests. Before creating another instance the manager checks whether a mediator instance has already been loaded into a memory. To speed up request dispatching the manager uses different heuristics algorithms that will be discussed later in the next chapter. To reach this goal the manager communicates

with mediators receiving necessary meta-information through the interface pre-defined by the XDA.

Therefore, the structure of mediator classes can be divided into two logical parts. One is fully determined by the layout of the persistent class being wrapped (and consequently by its IDL interface) and consists of the methods wrapping persistent class' member variables and functions. This is the most essential part of mediator classes. The second part is the functionality that is used by the object manager. This part has the structure common for all mediators, since it deals with mediator's constructors, destructors and some additional functions. Gathering this functionality together in form of base classes and macroses the XDA separates the management-specific functionality from the application-specific functionality of mediators. This separation and the usage of these pre-defined classes and macroses simplify the development of custom mediators.

The mediators are implemented as usual CORBA servants that additionally inherit XDA-specific functionality from the base class `XDA_ObjectBase`. For ordinal mediators the XDA provides ready-to-use macroses implementing all functions declared in the `XDA_ObjectBase`. If the mediators are not typical, for example, have parameterised constructors or define a view, then the programmer should provide a custom implementation for the corresponding from the base class `XDA_ObjectBase`. Please see Chapter 5 for more information about the implementation of mediators.

4.4.2.3 Request Processing

On the one hand, the introduction of intermediate mediator components enhances the flexibility and adaptability of the integrated system, as well as allows realising „object views“ on the structure of persistent objects. On the other hand, however, it brings a burden for mediator management. Our own contribution here consists in identifying the important components of the programming model for robust mediator management, pointing out design alternatives and efficient algorithms, collecting those components into a well-rounded execution model.

Therefore, the lifetime of mediators is managed by the XDA, which additionally takes on other responsibilities like handling transaction boundaries. However, it should be emphasised, that *the issues of mediator objects management are orthogonal to their internal structure*. Requirements to mediator management provided by object adapter can be summarised as follows.

- The adapter supports persistent CORBA objects implemented with the mediators approach (see Section 4.3.3).
- Only database objects are stored in persistent memory, where mediator objects are transient and allocated in dynamic memory.
- The adapter is responsible for dynamically instantiating and releasing transient mediator objects to persistent database objects, so that full persistent CORBA objects are available whenever they are needed.

Even though such „persistent CORBA objects“ are not fully kept in persistent memory, to the clients they appear as long-lived objects. Accordingly, this scheme is called *pseudopersistence* [RM97]. Every mediator object has a data member that specifies the set of database objects to which the mediator delegates operations. In this point, the mediators differ from regular TIEs. In a regular TIE, this data member is a usual C++ pointer or reference. In a mediator, it must be an ODBMS reference (`os_Reference`), since it points to a database object in persistent memory.

To allow on-demand activation of mediator objects, the adapter takes responsibility for the generation and interpretation of IPORs. When a mediator is instantiated, one must initialise its (*os_Reference*) data members. To support the instantiation of a mediator for a given persistent object reference (IPOR), the XDA provides necessary services for translating of the IPOR reference to corresponding database references. Figure 33 illustrates the activities occurring by request processing.

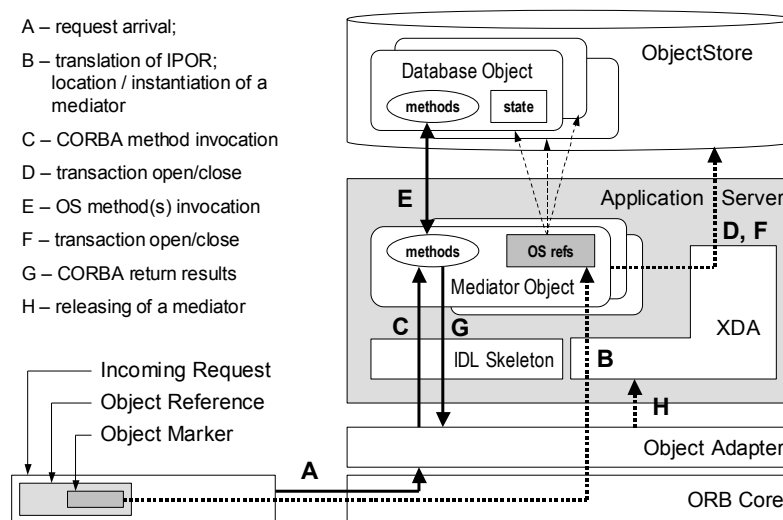


Figure 33. Mediator management

A request to a persistent object arrives through the ORB core (A), causing an upcall to an XDA-provided object activation function (B). The *object marker* field of the target object reference contains stringified database references to associated persistent objects. The XDA extracts the marker from the object reference and uses it for the location and initialisation of a compatible mediator object. The incoming CORBA request then reaches the target mediator object as an upcall through the IDL skeleton (C). Knowing the type of the transaction, if necessary, the mediator informs the XDA that an open transaction of a certain type is required for further invocation of OS methods (D). In turn, depending on the current transaction XDA transaction manager takes decision and guarantees the presence of a necessary transaction. Finishing all operation at OS database objects (E) the mediator informs the XDA that it does not need more the transaction and that it can be closed (F). Results of the called CORBA method are returned to the client as usual through the IDL skeleton (G). At the end of the operation, another upcall to the XDA (H) causes the target mediator to be released.

In Figure 33, the object activation upcall happens because the target of an incoming request is a mediator object that can be released. Pseudopersistence should be understood in the context of the multi-tier architecture in Figure 26 above. Persistent database objects do not live within an application server, but in a database server. Therefore, multiple application servers (middle-tier processes) may be clients of a database server; they may or may not run in the same node as the database server. Moreover, a persistent database object may be shared by multiple mediators, each in the address space of a different application server.

4.4.2.4 Transaction Management

Every act of reading or writing to the database must be within a transaction. Since in our approach we do not use OTS service, the XDA implements its own transaction control mechanism using conventional transactions provided by ObjectStore.

To eliminate defects of transaction management in existing systems presented in Section 3, a new model for transaction processing was developed. The model extends automatic transactions defined in ODAF by proposing new transaction's types and employing new algorithms for maximising their safety and performance. Additionally, representing database transactions as separate CORBA objects the XDA provides for CORBA clients the possibility to control transactions manually. Since ObjectStore supports serialisability only between separate UNIX processes, every CORBA client manipulates with databases through its own XDA application server.

Controlling Transactions with XDA

An XDA transaction represents a unit of work that is handled by the XDA's transaction manager and determined by the user or by the adapter. For the user the model assumes two basic interaction modes with the adapter: *automatic* and *manual*.

- *Automatic mode.* This mode assumes that the boundaries and types of transactions are controlled by the XDA automatically. Before the first request for an object is delivered, the manager automatically starts the corresponding transaction, if necessary, and activates the corresponding mediator. Automatic transactions are easy to use and primary intended for „pure“ CORBA clients that do not want to care about transaction's boundaries and modes.
- *Manual mode.* Manual transactions occur within boundaries that can be explicitly defined by the client, depending on runtime conditions. Manual transactions are more difficult to use but provide greater flexibility and control over database transactions. It should be also noted that the presence of this mode makes a cardinal difference between the XDA approach and other approaches like ODAF and OOSA, since they do not support this mode.

If a CORBA client does not care about transactions, by default an automatic mode is used. In this case, the XDA manages transactions automatically and guarantees that every access to persistent data will be supported by the right transaction. Using different heuristic algorithms and fuzzy logic the transaction manager tries to manage transaction in the way of gaining the maximum performance. Concrete algorithms and approaches for efficient automatic transaction management will be discussed later together with concrete XDA implementations.

The second mode is used if a CORBA client wants to control transactions manually. To give the clients a possibility to manipulate with database transactions, the XDA provides an API for controlling transactions. The functions are presented on two IDL interfaces: `XDA_Adapter` and `XDA_Transaction`. All general functions that are not specific for particular transactions such as `begin_transaction()`, `commit_current()`, `abort_current()` are available on the interface `XDA_Adapter`. Therefore, a client can use them directly after binding to the application server. Other functions that are specific for concrete transactions such as `commit()` and `abort()` are accessible through the interface `XDA_Transaction`.

The CORBA client uses the XDA's transaction control API as it used to do in databases. A transaction follows the following general flow.

```

XDA_Adapter_var aVar = ...;
XDA_Transaction_var trVar;

(1) aVar->begin_transaction(...);           // Begin Transaction
(2)  ...                                   // Do what needs to be done
(3)  if (...) trVar->abort();                // if error, abort
(4)  else trVar->commit();                  // else commit
(5)  if (trVar->is_committed()) ...;        // check status
(6)  trVar->destroy();                      // End Transaction

```

First, the beginning of the transaction should be indicated. This signals that whatever functions follow should be contained within the transaction. This procedure of identifying the beginning and ending of the transaction (also referred to as transactional boundaries) is known as *demarcation* (1, 5). It identifies also the type of transaction. A transaction in the XDA may be update or read-only, and either type of transaction can be nested within another transaction of the same type.

Once the actual instructions that we want to have controlled by the transaction are executed (2), the client can check to see if any errors have occurred. If they have, it can abort the transaction (3) rolling back any changes that have been made. Otherwise, it commits the transaction making the changes permanent (4). After finishing the transaction the client can check the status (committed or not) of the transaction (5). This step is optional. Then the client ends the transaction by deleting the actual transaction control object (6).

4.4.2.5 Transactional Models

There are many different types of and uses for transactions. How a transaction is used and how it is structured typically falls in to one of many transactional models. We will look at three models now supported by the actual XDA implementations, namely.

- Flat transactions
- Nested transactions
- Chained transactions

4.4.2.5.1 Flat Transactions

A single or atomic unit of work composed of one or more steps, the flat transaction is the simplest of transactions. Should one of the steps fail, the entire transaction is rolled back. Flat transactions are the de facto standard for all database operations.

They appear in the manual mode as transactions explicitly defined by the client or in the automatic mode if the „transaction per operation“ policy is used.

4.4.2.5.2 Nested Transactions

The nested transactions model has atomic units embedded in other transactions. This has the effect of a transaction „tree“ – a root transaction that contains several branches of other transactions. The nested transaction differs from the flat transaction, in that should one of the sub-transaction roll back, it will not affect the parent transactions. In other words, the failure of a transaction is limited to just that transaction. A sub-transaction can be either a flat transaction or another set of nested transactions. The following sequence diagram illustrates using nested transactions (Figure 34). After transaction A begins, it starts transaction B. Transaction B, in turn, invokes first transaction C, then transaction D.

The nested transactions appear only in the manual mode as transactions explicitly defined by the client.

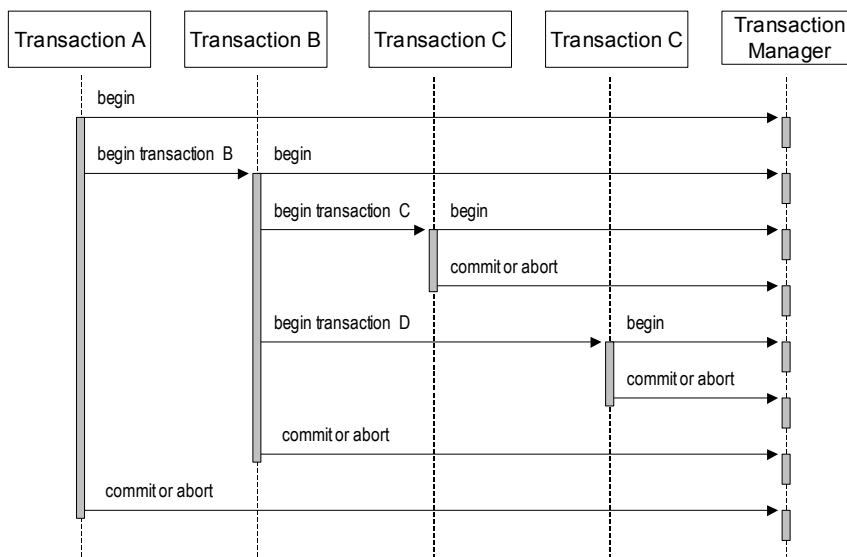


Figure 34. An example of the nested transactions

4.4.2.5.3 Chained Transactions

Chained transactions, sometimes referred to as *serial transactions*, are a set of contiguous transactions related together. Each transaction relies on the results of the previous transaction. With chained transactions, when the first transaction commits, its resources (say, locks) are retained and immediately made available to the next transaction in the chain – effectively combining the `commit()` and the next `begin_transaction()` into a single atomic step using an operation `checkpoint()`. The result is that transactions outside of the chain cannot see or alter the data being affected by the transactions in the chain. This is different from committing a transaction and then starting a new one in two separate steps, as that would result in the resources used by the first transaction being released.

The basic advantage of the chained transactions is that resources (such as locks), used by one transaction and that are required by subsequent transactions, are kept. This is more efficient than sequential releasing and achieving of the resources for every transaction in the chain. However, if one of the transactions should fail, only the currently executing transaction (the one that failed) will be rolled back – the rest of the previously committed transactions will not. Another drawback of chained transactions is that they can potentially lock up a lot of valuable database resources, thus slowing the system down. Careful thought should be given when using these transactions.

Typically, they appear in the automatic mode if the „delayed transactions“ policy is used. The XDA identifies the boundaries of each transaction and then submits all of the transactions as a group.

4.4.3 Developing integrated CORBA/ODBMS Applications

Developing typical CORBA applications the first thing to do is to define IDL interfaces. For CORBA/ODBMS applications it is not so. Since database applications can (and frequently do) operate autonomously, the definition of IDL interfaces is a secondary step, which depends on actually existing persistent data. In the first case, an IDL interface serves as a key element defining the layout of the classes, which will be implemented in the application, whereas in the second case the layout of persistent classes is already defined by the database. The question is, which of them - IDL interface or persistent data determines the sequence of the application development path (or which is primary).

When IDL is an entry point to the application's development, the natural solution is to implement mediator C++ classes, and subsequently define persistent-capable database C++ classes, to which the delegations of the mediators are directed. However, this approach does not work if the persistent classes already exist and/or are used by other applications. In this case, the programmer has to implement an IDL interface through additional transient objects delegating all requests to their persistent counterparts, and avoiding changes in the database schema. In turn, such a back mapping is rather difficult because of the inherent differences between IDL, which is an interface definition language, and implementation languages like C++²⁹. Anyway, it is not impossible. Now let us see typical phases of the application development process in each case.

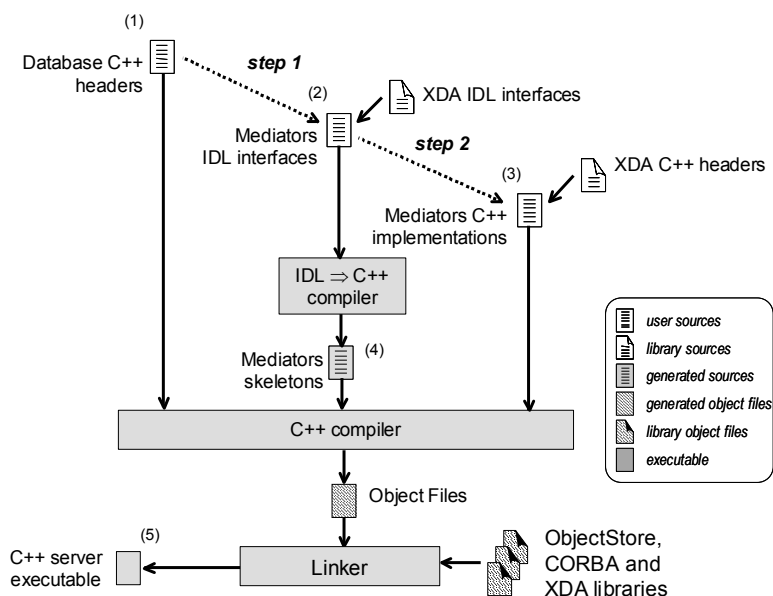


Figure 35. CORBA→ODBMS application development steps

4.4.3.1 CORBA→ODBMS Application

The process of CORBA/ODBMS application development takes two steps. The implementation of the CORBA application, which uses ODBMS as its backing store, usually adopts the following scheme (Figure 35). Such a process typically begins from the definition of the IDL interfaces (1), then implementation of necessary persistent database classes (2) and corresponding mediator classes (3) that are attached to the automatically generated skeletons (4). Finally, C++ compiler and linker make the CORBA server executable, which is simultaneously the ODBMS client (5).

The compilation and link process of CORBA clients are quite conventional. Created in the first step IDL interface specifications are used to generate CORBA proxies through which a CORBA client communicates with the application server. In the contrast to the client that uses only CORBA libraries, the application server must be linked with additional object files and libraries. Thus, since the application server is also an ObjectStore client, it must be compiled and linked in compliance with the same rules as regular ObjectStore clients. Moreover, it must also be linked with the XDA library. Therefore, the process of building CORBA-based distributed database applications facilitated by the XDA requires the full compliance of all these libraries.

²⁹ Actually, it is almost impossible, at least because due to the current mapping of IDL to C++.

4.4.3.2 ODBMS→CORBA Application

In the second case, where an existing database application uses CORBA environment to extend its heterogeneity, distribution and flexibility, another schema for the implementation of the integrated CORBA/ODBMS application must be used (Figure 36). It is assumed that the database classes already exist³⁰ (1). In the first step the programmer defines mediators IDL interfaces (2) mapping the original database C++ classes to mediators. In the second step the defined IDL interfaces must be implemented in C++ (3). Subsequent steps for the compilation and linkage with object files, CORBA, ODBMS and XDA libraries are the same as in the previous case.

It should be emphasised, that the properties of wrapped database application can be dynamically changed: until the IDL interface persists, the CORBA database clients can operate without modifications. Moreover, it is not necessary to recompile the CORBA server, until the database schema remains unchanged. This gives a great degree of flexibility of the overall CORBA/ODBMS system. Hence, returning to the discussion from the Chapter 4.3.3, the same conclusion can be drawn: the mediators-based solution is the best choice for the integration of database applications with the CORBA environment. Separating persistent and CORBA-capable classes in two completely independent groups, this method provides a natural compromise between the system’s complexity and flexibility.

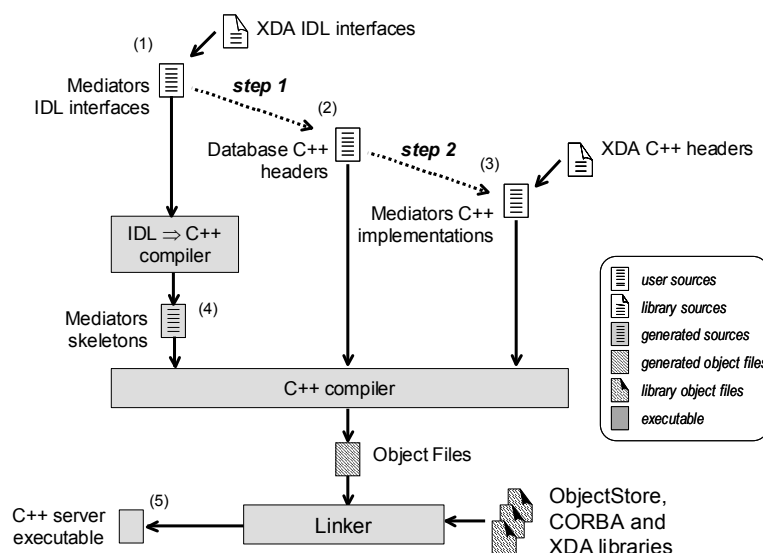


Figure 36. ODBMS→CORBA application development steps

4.4.4 Comparing XDA-based and native ObjectStore Applications

This section makes a comparison between native database applications and CORBA-capable database applications trying to highlight their benefits and drawbacks.

4.4.4.1 Native ObjectStore Application

A typical database application uses ObjectStore’s API to handle transactions, manipulate collections, references, roots and other specific ObjectStore data structures. Database clients are heavyweight, tightly coupled with the server (i.e. with the structure of data stored in database), and can be implemented exclusively through a limited number of programming

³⁰ Moreover, very often the already-functioning database application was created without any knowledge about future CORBA enhancements.

languages supported by ObjectStore. Regarding these features, ObjectStore clients can be referenced as low-level database clients.

The objects (more precisely, their states) are transferred to clients side completely. Since the implementations of class member functions are provided by ObjectStore client applications, the clients are free to manipulate data in arbitrary way. In fact, only trusted clients can work in distributed ObjectStore system. If we decide to provide alternative database clients which.

- would have limited access to persistent objects' members,
- could not provide own method implementations,
- would not be dependent on database schema,
- would be implemented through programming languages other than the ones supported by ObjectStore,
- would be lightweight,

then we should look for other solutions, for example, for an integrated CORBA/ODBMS solution.

4.4.4.2 XDA-based CORBA/ODBMS Application

In the CORBA/ODBMS environment, the clients interact with the database using CORBA. The client is loosely coupled with the database server either from logical, or from implementation perspectives. Because of these features CORBA database clients can be regarded as high-level database clients.

Let us imagine that in our distributed application we need a database client, which is permitted to perform only a subset of operations defined in ObjectStore classes. This arrangement can be easily done through the mediators (see Section 4.3.3): the unnecessary methods can be simply not declared in the IDL interface.

In addition, to avoid exhibition of fine-grained interfaces to the distributed environment, we can provide several mappings, for example the mapping by value (Section 4.3.2.9). The CORBA database clients will then manipulate value-based representation of objects instead of references to remote objects. That will result in a considerable improvement of the system's performance and clarify CORBA database clients' perspective.

Obviously, CORBA database clients will have only limited access to members of persistent objects, thus corrupt clients will not threaten the data stored in the database. Naturally, the mediators can leave the structure of persistent classes untouched, thus giving CORBA database clients the same „image“ of the persistent objects as regular database clients have. Still, even in that case, we can manually add to the default mediator functionality additional functions checking whether a particular CORBA database client is authorised to access certain operations on persistent objects. It is an example showing the possibility to add specific functionality to the mediators' code.

4.4.4.3 Summary

As the result of the integration, the database is accessible not only by regular database clients, but also by CORBA database clients. This section has shown that the use of components implemented in the XDA has many benefits over the static systems described in Chapter 3. Some of them are the following.

- *Transparent persistence.* The proposed model preserves *orthogonal persistence*, if it is supported by the underlying ODBMS. When the persistence of the objects is

determined during their instantiation, which is offered by the most ODBMSs, only *transparent persistence* is supported.

- *Extensibility*. Pluggable components allow an easy adoption of new technologies and the integration of third party implementations into the XDA framework.
- *Flexibility*. Developers and users can choose at runtime which components and component technologies best suit to their requirements.
- *Easy adoption*. Programmers can adopt new technologies by just reimplementing one or two adapter classes such as the `XDA_TransactionManager` and the `XDA_ObjectManager`.

However, on the one hand the pluggable components provide great facilities for systems' extensibility and, therefore, guarantee that the XDA core can be extended and upgraded for future technologies. On the other hand, during the integration of third-party technologies some drawbacks also have to be considered.

- *Partial use of the new technology*. It can happen that through the integration of the system some features of this technology can be lost or cannot be used. For example, a DBMS provides its own transfer protocol for messages over the network so that clients on remote locations can talk directly to the servers. To have a clear communication model this features will not be used in XDA.
- *Performance loss*. Usage of mediators and CORBA-compatible representations for persistent objects adds overhead to the system.

The summary of comparison of features facilitated by CORBA database clients and regular database clients is presented in Table 14. Further, Chapter 7 provides a comprehensive comparison for performance of XDA-based and native ObjectStore applications.

	XDA-based	Native ObjectStore
Used interfaces	IDL interfaces of the XDA and database-specific Mediators	C++ ObjectStore API
Transaction handling	Implicit	Explicit
Performance	Average	High
Product	Any IIOP-compliant CORBA tool	ObjectStore only
Programming language	Any programming language the IDL compiler supports (C++, C, Smalltalk, Java, Ada95, COBOL, PERL, etc.)	C++, C, Java, ActiveX ³¹
Coupled with the server via	CORBA's IIOP	ObjectStore native page-based communication
Other properties	lightweight does not define methods' implementations	heavyweight defines methods' implementations

Table 14. Comparison between features of database and CORBA clients

³¹ In ObjectStore the access from Java and ActiveX clients to C++ objects is based on the proxy pattern, i.e. by means of instances of specially generated C++ proxy classes. Still, there are numerous restrictions concerning the proxies. For example, proxies for Java use JNI, so there is no way to use them from inside an applet.

Chapter 5

Implementation of the XDA Framework

In order to prove the viability of the proposed method and to investigate the issues of the CORBA/ODBMS integration in practice, a first working prototype of the eXtensible Database Adapter (XDA) has been developed. The prototype is conceptualised, designed and implemented for a concrete commercial ODBMS ObjectStore [ODIug01].

The first prototype of the XDA has been designed and implemented to verify our model and to confirm its viability. In fact, using the developed prototype in some research projects to access real geospatial databases [SB00], we have identified the crucial problems occurred during the integration process and evaluated the applicability of such a tool [SC00]. The results received during this evaluation were used later for development of the next version of the adapter.

The following sections present the concepts, design of the first version of the XDA, and discuss its advantages and disadvantages. The first section presents the architecture and implementation concepts of the first XDA prototype that are used in the following sections where the design details are discussed. Finally, the drawbacks and critical issues for performance of the integrated CORBA/ODBMS system are outlined.

5.1 The XDA Architecture

The first prototype of the XDA was developed as an extension to the standard *Basic Object Adapter (BOA)* (see Section 2.2.6). *Orbix* from IONA Tech. [ITorbix97] was used as the base implementation of CORBA [OMGorb97].

In the above presented component-based architecture of the XDA, the framework was realised with the help of independent components – *managers* (see Section 4.4.2). According to this basic design solution, the core of the adapter consists of the main container module *XDA_Adapter* and four replaceable plug-in components – managers: a *loader* that is an equivalent of the *object manager*, a *database manager*, a *type manager* and a *transaction manager*. The following Figure 37 presents the class diagram of the XDA prototype.

The *loader* (or *object manager*) is a manager, which is responsible for processing incoming requests. It retrieves persistent objects and creates new mediators when necessary. The manager is implemented as an instance of the *XDA_Loader* class that is a specialisation of the Orbix-specific pre-defined *CORBA::LoaderClass*. In the time of this work the loading was not a standard feature available in every ORB, but an extension that was specific for *Orbix* (see Section 2.2.8.1 for more details about loading). Therefore, the presence of this feature was one of the reasons for choosing *Orbix* as an implementation platform.

The *database manager* is an instance of the *XDA_DatabaseManager* class. The manager keeps track of all mediators and controls the segmentation of persistent objects in the database.

The *type manager* is an instance of the *XDA_TypeManager* class. It administers meta-information about the whole set of all available mediator classes and their relations to persistent counterparts.

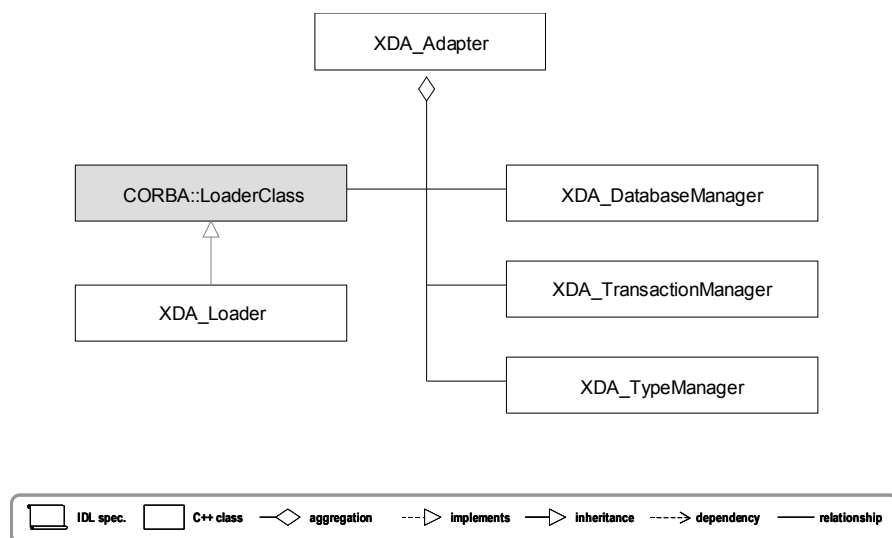


Figure 37. The XDA class diagram

The *transaction manager* is an instance of the `XDA_TransactionManager` class. It controls the start and the end of transactions.

The `XDA_Adapter` class represents the public facade for all managers, controls them and allows their replacement. This simplifies the use of the XDA.

5.2 Implementing Mediator Components

Implementing a custom wrapper on top of the XDA framework, the programmer can use default managers and should implement only mediator classes corresponding to database classes.

As it was mentioned, earlier original database objects are made accessible for the CORBA environment with the help of the corresponding *transient* CORBA objects – *mediators*, which act as local database clients. For each original ObjectStore C++ class, a server programmer should create the corresponding CORBA C++ mediator class with an equivalent IDL interface compliant to the defined C++ to IDL mapping (see Section 4.3.2). In this interface, the original data structures from the ObjectStore C++ data model should be represented by appropriate structures in the CORBA IDL model. Because of differences in the object's representations in both models automatic mediator's code generation from the specifications of corresponding database classes is possible only in particular cases (see Section 6.2.3). Therefore, a user-defined implementation of mediators is assumed. To make implementation of mediators easier, the XDA provides a special template class `XDA_ObjectImpl` that implements database specific functionality. The template should be inherited by every mediator and tuned to the corresponding database class. The complete schema of relations between these classes is shown in Figure 38.

The main idea of the template is to keep the specifications of mediators classes free from the XDA-related behaviour and concentrate it in the template. This makes a clear separation between the functionality of the mediators that is related to the XDA and the functionality that is related to the mediation. Additionally, the template implements the operations from the `XDA_Object` IDL interface that are common to all mediator objects and thus helps the programmer to speed up the whole development process.

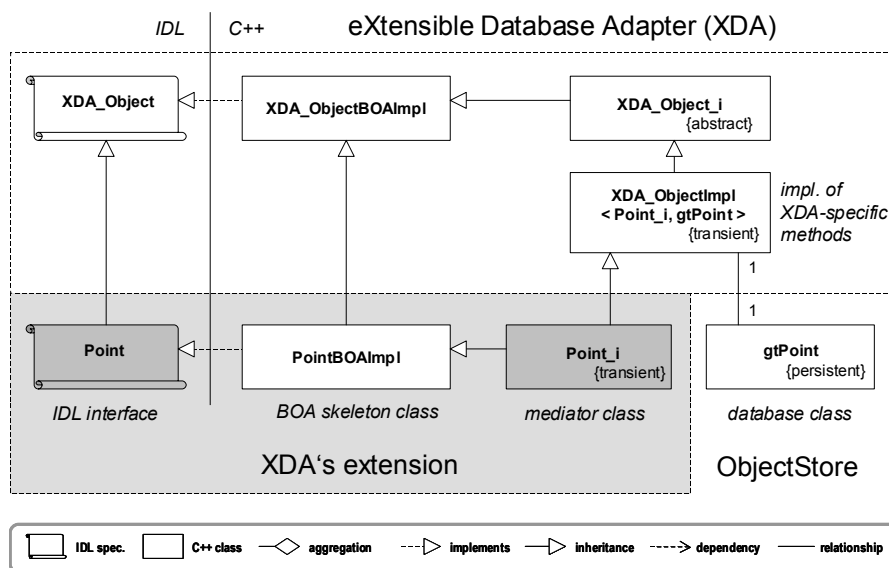


Figure 38. Implementation of the user-defined mediator

The transience of mediators not only saves database space, but also provides a complete separation between the original database and communication components. This separation allows working with the database not only through the new CORBA interface, but also through the original C++ interface. Moreover, when employing this method any already existing data store with a compatible schema can always be used through the CORBA-interface. Referential integrity of the data shared by local and CORBA-conformant applications is guaranteed by the use of the native ObjectStore transaction control mechanism.

5.3 Request Dispatching and Mediator Management

The presence of the target being non-compatible with CORBA database objects and the transient mediators forces the XDA to take the responsibility for dispatching corresponding in-/out-coming requests. For these purposes, the XDA extends the functionality of the standard object adapter for the generation and processing of object references, request dispatching and activation/deactivation of mediators.

5.3.1 Interoperable Persistent Object References

CORBA clients work with remote mediator objects through *Interoperable Persistent Object References (IPORs)* that identify remote objects in the CORBA environment. For the identification of database objects the XDA modifies the usual IOR references saving additional information in the *Object ID* (see Section 4.3.3.3). This information depends on the concrete mediator's type. For unique identification of mediators and corresponding database objects it is sufficient to know the type name of the mediator, which is save in *Class ID*, and the corresponding local database pointer (Persistent Object ID, Figure 39). Actually, it could be a sequence of persistence pointers, since one object can be implemented with the help of several persistent database objects.

This format of the IOR is quite powerful and additionally allows the XDA, besides providing persistence to CORBA objects, to provide persistence to the corresponding object references.

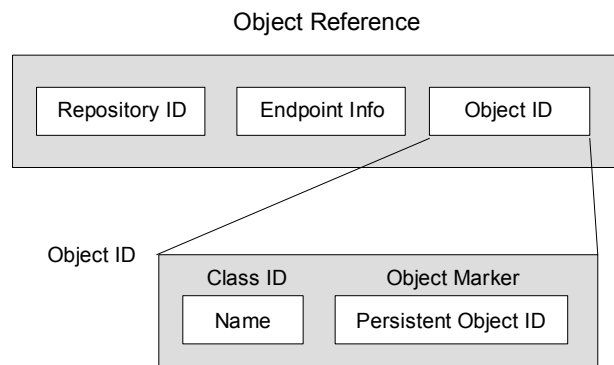


Figure 39. Implementation of IPOR

5.3.2 Activation

IPORs identify remote persistent objects in the CORBA environment. However, the request to a persistent object can be processed only by a corresponding transient mediator. If a mediator was previously deactivated, than the XDA’s loader is asked by the ORB runtime system to retrieve the missed mediator. Figure 40 presents the full schema of request dispatching.

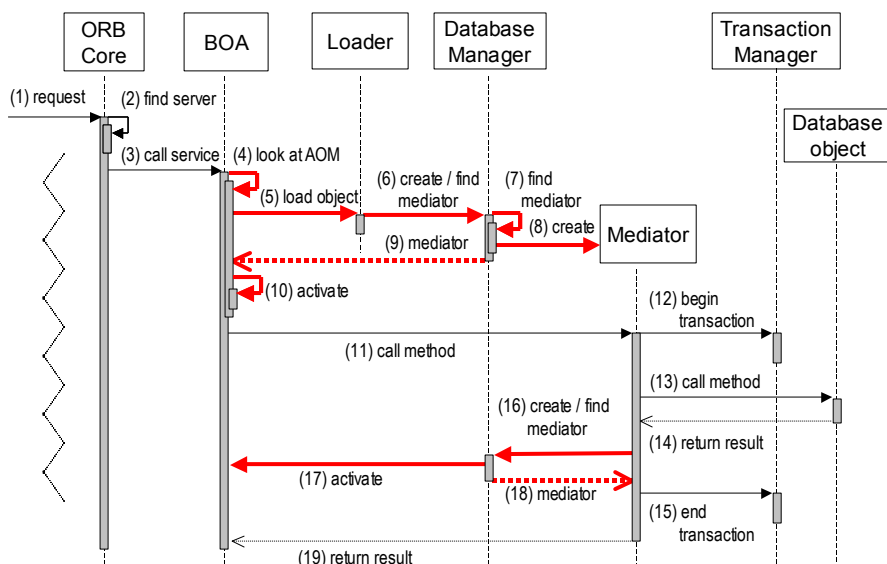


Figure 40. Request dispatch in the BOA-based XDA

Here it is simple to see that the basic component that maintains the binding between mediators and database objects is the *database manager*. Arrows corresponding to the process of reference resolving are marked in bold (Figure 40). The database manager is used here not only by the loader for resolving the incoming IPORs, but also by mediator objects for creation of returning IPORs. Since some methods of database classes receive/return as result references to other database objects. CORBA compatible methods of the mediators must receive/return IPOR references to the corresponding mediator objects. Therefore, the database manager controls the binding between database objects and mediators providing service functions for obtaining a mediator for given database objects and back.

For example, the database manager provides a function that takes an incoming IPOR CORBA reference and converts it to the corresponding local ObjectStore reference to database objects. Here the manager cares about right correspondence of mediators to their database objects, e.g. to avoid the creation of multiple transient mediators for the same

persistent database object. It first checks, if a mediator to the first associated database object already exists in the server's address space, then it returns a duplicated reference to either an existing mediator or a newly instantiated one. Later in this section an example of reference resolving for input/output parameters in a mediator is presented (Figure 43).

A straightforward implementation of the functionality provided by the database manager would be to use the global ORB table of available objects. A non-standard `bind()` function, present in various ORBs, could be used to perform the check mentioned above. Given a IPOR reference, one would convert it to string, which would then be passed as an argument to `bind()`. Then the ORB will find the corresponding mediator object or invoke a loader that will make the new one. However, this simple strategy has two drawbacks. First, this method does not provide the functions supporting back mapping of database references to the corresponding CORBA IPOR references. The second drawback is low search speed and since the `bind()` always increases the object's reference counter, it is not simple to determine the correct counter's value in the case of multiple clients.

Therefore, another more efficient method was implemented in XDA. Instead of using ORB tables, it keeps pairs (database reference, CORBA IPOR reference) in its own tables of active mediators, which it hashes by database references with a hash function provided by the ODBMS. The associations are stored in the special transient dictionaries of references, which are organised in the hierarchy according to object's locations (Figure 41).

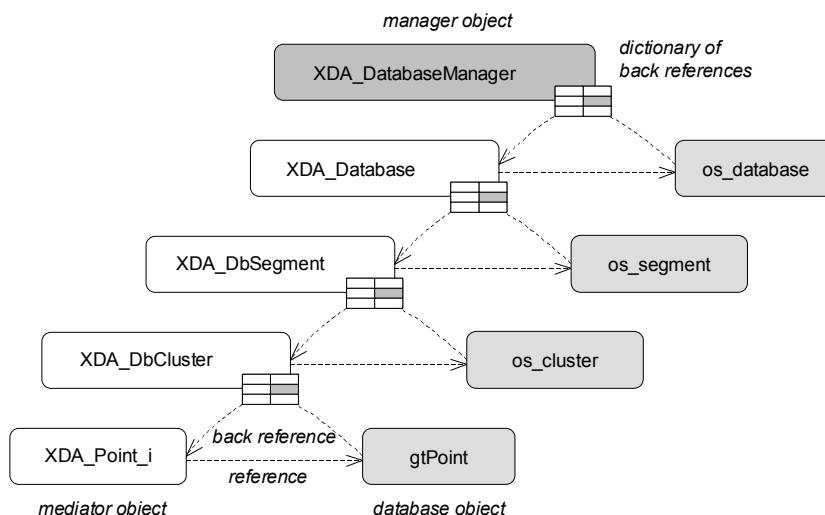


Figure 41. Hierarchical organisation of dictionaries for back references

Using a reference to a database object as the key it is possible to find a reference to the corresponding mediator more effective than using usual CORBA objects. The schema of request dispatching for persistent objects in the XDA is presented in Figure 42. The uniqueness of pointers and the use of hash tables make the retrieval from the dictionary very efficient. Moreover, the XDA reduces the search organizing dictionaries in the hierarchy of small dictionaries according to object's locations. During the search, the manager finds the database and the segment to which the database object belongs and goes through the hierarchical organisation of the dictionaries to the target mediator. If a programmer knows to which ObjectStore database or segment the resulting database object belongs, he can accelerate the search procedure directly calling the corresponding mediator (of the database or the segment).

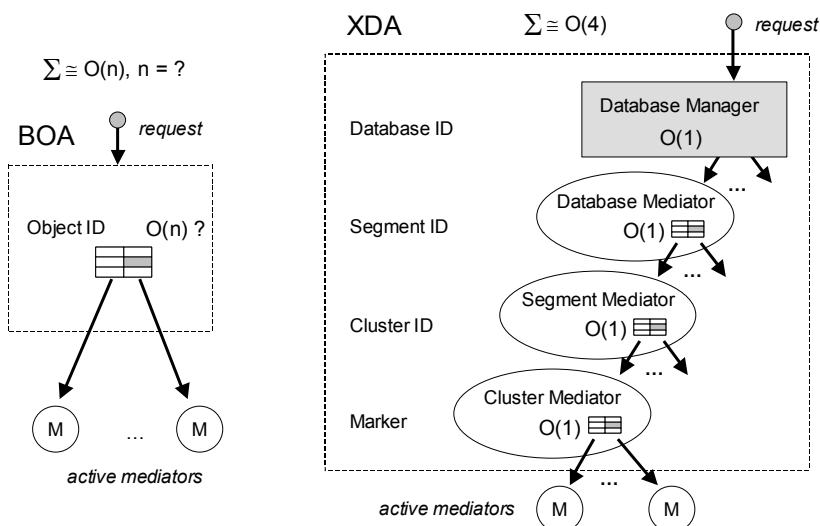


Figure 42. Request dispatching for persistent objects

Figure 43 presents an example of reference resolving for input/output parameters in a mediator. Here request processing performs in the following way. The BOA automatically translates all input IPORs in local C++ references to corresponding mediators looking for an object in his table of active objects. This means that if a corresponding mediator object is deactivated, it will be not found there. In this case, the XDA loader will be implicitly invoked for the instantiation of new mediators. Following the example presented in the Figure 43, the conversion of an incoming IPOR CORBA reference to the corresponding local ObjectStore references to database objects is performed in two steps.

```

GTA_BoundingBox_ptr
GTA_BoundingBox_i::get_intersection (GTA_BoundingBox_ptr bbPtr)
    throw (CORBA::SystemException)
{
(1) XDA_Object_i * objImp = (XDA_Object_i *) bbPtr->deref();
    GTA_BoundingBox_i * bbImp = (GTA_BoundingBox_i *) objImp->imp_deref();
    GTA_BoundingBox_i * bbNewImp;
    gtBoundingBox *bbOs, *bbNewOs;

    XDA_Transaction_i * trImp = _xda()->transaction_manager()->begin
        (XDA_Transaction::update, XDA_Transaction::server);
    TIX_HANDLE (err_objectstore) {
(2)     bbOs = (gtBoundingBox*) bbImp->os_deref();
        bbNewOs = &(os_ref()->intersection (*bbOs));
    } TIX_EXCEPTION {
        XDA_EXCEPTION (type_name(), "get_intersection()", \
            "ObjectStore error:" \
            << tix_handler::get_current()->get_report0());
    } TIX_END_HANDLE
    trImp->commit_destroy();

(3) bbNewImp = (GTA_BoundingBox_i *) GTA_BoundingBox_i::get(bbNewOs)->imp_deref();
    return GTA_BoundingBox::_duplicate(bbNewImp);
}

```

Figure 43. Reference resolving for input/return parameters in a mediator

First, the incoming bbPtr reference will be downcasted to the reference to the corresponding mediator implementation class (1) and, second, a virtual function `_os_deref()` is invoked returning a reference to the associated database object bbOs (2).

Back conversion of ObjectStore references to CORBA references is performed with the help of a function `get ()` (3). Since in the most cases the class of the corresponding mediator for returned ObjectStore references is known, this function is implemented as a static function on mediator classes.

5.3.3 Deactivation

Another important function of the database manager is the *deactivation* of unused mediators. Mediators that are no longer needed should be *temporarily* removed from the server's address space to avoid memory overloading. However, releasing all mediators at the end of every operation appears also unreasonable, since the XDA only needs to ensure that these mediators will be eventually released. Postponing their destruction would avoid the costs of later instantiations.

Caching mediators makes sense if the references to used persistent objects remain valid across transactions. So far, we have ignored database transactions, this topic will be discussed in the following Section 5.4. Let us assume, by now, that transactions are started and committed (or aborted), and that each operation is encompassed by an individual transaction. To keep the references to be valid across transactions, `os_Reference` class was used for the implementation of references to persistent objects.

The BOA-based XDA realised a simple LRU algorithm for caching of mediators that is based on using the maximum number of active mediators. The adapter actually caches the last N mediators it instantiated, where N is a configurable parameter. At the end of every operation the XDA brings the number of mediators down to $N+\delta$, keeping the most recent ones. The δ accounts for any duplicate calls that might have been issued by the server code.

```

// connect to the server and open a database
GTA_Database_var dbVar = ...;

try {
    // create an object in the default segment of the database
    GTA_Point_var p1Var = GTA_Point::_narrow (dbVar->create_object("Point"));

    // create object in the new segment
    GTA_Segment_var segVar = dbVar->create_segment ();
    GTA_Point_var p2Var = GTA_Point::_narrow (segVar->create_object("Point"));

    ...
(1) p1Var->evict (); // explicitly delete transient mediator for one object
(2) segVar->evict (); // for all objects in the segment
    ...
    p1Var->set_Double (250000, 560000, -1000);
    // implicit loading of mediator object
    ...
    p2Var->destroy (); // destroy persistent object
    segVar->destroy (); // further usage of the object is not possible
    dbVar->close ();
    ...
} catch (...) {
    ...
}

```

Figure 44. Example of the CORBA client code that performs explicit deactivation of mediators

With caching, CORBA clients have also a way of forcing the removal of objects from the cache explicitly. For these purposes, the XDA provides a set of functions defined on its IDL interface through which CORBA clients can notify the server that particular mediator objects are no longer needed. Since the dictionaries of the back-references (Figure 41) are reused as tables of the active mediators, the mediators are cached according to their placement in the

database. Using this structure the XDA prototype provides functions for deactivation of mediators. An example of their usage is presented in the Figure 44. These functions allow a CORBA client to explicitly deactivate a particular mediator (Figure 44, 1), all mediators for objects located in a particular segment (Figure 44, 2) or all mediators for objects in a particular database.

5.4 Transaction Management

Before the first request for an object is delivered, the XDA automatically activates the corresponding mediator and starts a corresponding transaction, if necessary.

5.4.1 Demarcation and Management of Transactions in XDA

The process of determining where a transaction begins and ends is called *demarcation*. For the basic description of transaction management see Section 4.4.2.4. Here we will describe details of demarcation and its differences for server and client programmers.

The CORBA client can manipulate with transaction boundaries using the corresponding methods from the XDA_Adapter and XDA_Transaction IDL interfaces. They allow the client to begin, commit or abort a transaction explicitly when it is necessary. The XDA also allows the client to nest transactions and support all transactional modes that ObjectStore provides: *read_only* and *update* transactions. For example, to start a new transaction, a function `begin_transaction()` on the XDA_Adapter interface is used.

```
XDA_Adapter_var xda;
XDA_Transaction_i * txn = xda->begin_transaction
(XDA_Transaction::read_only);
```

The server programmer can manipulate transactions with the help of functions implemented on the C++ interface of the XDA transaction manager. This interface allows the server programmer to perform the same operations as available for the CORBA client and a little bit more. For example, to start a new transaction a function `begin()` on the interface of the XDA transaction manager is used.

```
XDA_Transaction_i * txn = _xda()->transaction_manager()->begin
(XDA_Transaction::read_only);
```

In the XDA there are two methods of transaction demarcation – *automatic* and *manual* respectively to the two basic transaction processing modes defined. Therefore, if the programmer manages the transactions *manually*, he/she uses *programmatically demarcations*. Programmatic demarcations force the transaction manager to begin a real transaction unconditionally. This method is typically used by CORBA client and server programmers, since they want to control physical transactions. To begin such a transaction an additional parameter `XDA_Transaction::manual` should be provided. For example, for the CORBA client and server respectively.

```
XDA_Adapter_var xda;
XDA_Transaction_i * txn = xda->begin_transaction
(XDA_Transaction::read_only);

XDA_Transaction_i * txn = _xda()->transaction_manager()->begin
(XDA_Transaction::read_only, XDA_Transaction::manual);
```

Declarative demarcations are used only by the server programmer implementing mediator's functionality to define the boundaries of necessary transactions. If the programmer decides that a particular operation needs a particular transaction, he/she uses a *declarative demarcation*.

Actually, declarative demarcations differ from programmatic in where the boundaries are placed. With programmatic demarcation, the programmer determines exactly where the transaction will begin and end. With declarative demarcation, the programmer also determines the boundaries and type of the transaction, but this information will be used by the XDA transaction manager in *automatic* mode. The declarations give to the transaction manager necessary information about where and which transaction is required. It does not mean that exactly in this place a transaction must be started and closed. Trying to win some performance, the transaction manager usually processes several declarative transactions within one physical transaction. Therefore, declarative demarcation is a traditional method that is used by the server programmers implementing mediators functionality. For example, to begin a declarative transaction an additional parameter `XDA_Transaction::automatic` should be provided. The following Figure 45 presents an example of the CORBA server code that performs the demarcation of a declarative transaction of transaction's type „update“. Here the first invocation of the function `begin()` marks the begin of the transaction (1) and the second invocation to a function `commit_destroy()` closes the boundary of this transaction (2).

```
GTA_BoundingBox_ptr
GTA_BoundingBox_i::get_intersection (GTA_BoundingBox_ptr bbPtr)
    throw (CORBA::SystemException)
{
    XDA_Object_i * objImp = (XDA_Object_i *) bbPtr->_deref();
    GTA_BoundingBox_i * bb2Imp = (GTA_BoundingBox_i *) objImp->_imp_deref();
    GTA_BoundingBox_i * bbNewImp;
    gtBoundingBox *bb2Os, *bbNewOs;

(1)  XDA_Transaction_i * trImp = _xda()->transaction_manager()->begin
      (XDA_Transaction::update, XDA_Transaction::automatic);
      TIX_HANDLE (err_objectstore) {
          bb2Os = (gtBoundingBox*) bb2Imp->_os_deref();
          bbNewOs = &(os_ref()->intersection (*bb2Os));
      } TIX_EXCEPTION {
          XDA_EXCEPTION (type_name(), "get_intersection()", \
                        "ObjectStore error:" \
                        << tix_handler::get_current()->get_report0());
      } TIX_END_HANDLE
(2)  trImp->commit_destroy();

    bbNewImp = (GTA_BoundingBox_i *) GTA_BoundingBox_i::get(bbNewOs)->_imp_deref();
    return GTA_BoundingBox::_duplicate(bbNewImp);
}
```

Figure 45. Example of the CORBA server code that performs the demarcation of a declarative „update“ transaction

5.4.2 Manual Mode

The most fundamental decision that the XDA client programmer has to take is whether to manage the transaction manually or to let the XDA do the transaction management. In the manual mode, the responsibilities for transaction management are split between the XDA transaction manager and the client. Usually after the client's explicit start of a new transaction the XDA switches off the automatic transaction control delegating this responsibility to the client. However, the client is able to switch it on later. For example, it might be useful in the situation where all nested operation calls should be done within separate sub-transactions.

Controlling transactions manually the programmer can write rather complex applications, which logic from the beginning assumes the presence of some incorrect situations, to process them correctly. The standard behaviour of the transaction manager in the automatic mode

makes impossible the fine processing of possible abnormal situations that can occur in transactions using declarative demarcation. In this case, the usage of programmatic demarcation is recommended.

However, the programmatic demarcation can be a potential source of problems. Using the manual mode the programmer implicitly switches off the automatic control for available resources and takes this responsibility on his/her own. If a method invokes a long-running process (like a large loop), it blocks the other transactions from getting access to the data used in the method. Even if the loop might seem small, in concurrent systems that small delays can grow to large bottlenecks very quickly. The smallest gains can have performance improvements of an order of magnitude. Therefore, if an IDL operation has a slow process within it, the mediator's programmer should consider isolating the database calls manually or using the automatic mode. Either way, the programmer has a great deal of flexibility available to him.

Locking and Nesting Issues

For a number of reasons, it is useful to allow manual transactions to be nested. For example, suppose one transaction is required to hide intermediate results. This also allows the rollback of persistent data to its state as of the beginning of the transaction. However, suppose you would like to be able to roll back persistent data to its state as of some point after the beginning of this transaction. To allow this, the CORBA client can use a nested transaction that starts at this later point.

In addition, allowing nested transactions means that a routine that initiates a transaction can be called from both inside and outside a transaction. When a nested transaction is aborted, persistent data is rolled back to its state as of the beginning of that nested transaction. However, additional care should be taken here by reassessing the same data. Locks acquired during a given nested transaction are released upon whichever of these conditions comes first.

- Abort of the given nested transaction
- Abort of a transaction within which the given transaction is nested
- Commit of the top-level transaction within which the given transaction is nested

This means that after a nested transaction aborts, the client should not use persistent values retrieved during the nested transaction, because they might not be consistent with values retrieved later in the same top-level transaction. Consider, for example, the following nested transactions.

```
start top tx1
  start nested tx2
    read persistent var x
  abort nested tx2
  start nested tx3
    read persistent var x
  commit nested tx3
commit top tx1
```

The two read operations performed on x might not have the same result. Another client can change the value of x in the interval between the abort of the first nested transaction and the next read of x .

5.4.3 Automatic Mode

The automatic transaction control mode presumes that a client does not control transactions. In this case, the transaction manager tries to manage transactions optimally using declarative

transaction demarcations in mediators. Before a program can access persistent data, the transaction manager must start a transaction. In other words, the transaction manager guarantees that access to persistent data will always take place within a suitable transaction. But that does not mean that a new transaction will be started. For example, the transient objects do not need any transaction. Therefore, to find optimal transaction boundaries, the transaction manager uses different heuristic techniques.

The server programmers define the types and mark the beginning and the end of transactions in mediators by using function calls provided by the XDA's transaction manager. In the following Figure 46 arrows in bold show where a mediator invokes the transaction manager during request processing. The mediator requests the presence of a suitable transaction invoking a function `begin()` on the XDA transaction manager that returns a pointer to an object `XDA_Transaction` that represents the *virtual transaction* of the current session. This separation between real and virtual transactions allows the XDA transaction manager dynamically to define transaction boundaries, depending on runtime conditions. By default, the manager performs a „conservative“ method for transaction management. Basic rules of this method are the following.

- do not start a new transaction, if it is not necessary
- to keep an open transaction as long as possible
- to use the previously opened transaction, if possible
- to prefer „read_only“ to „update“ transactions, if possible

For example, the transient objects do not need any transaction. The manager always tries to reduce the frequency of transaction starts trying to use the previously opened transaction, if possible. If the types of the transactions are different, the first one will be closed and a new transaction of the given type will be started.

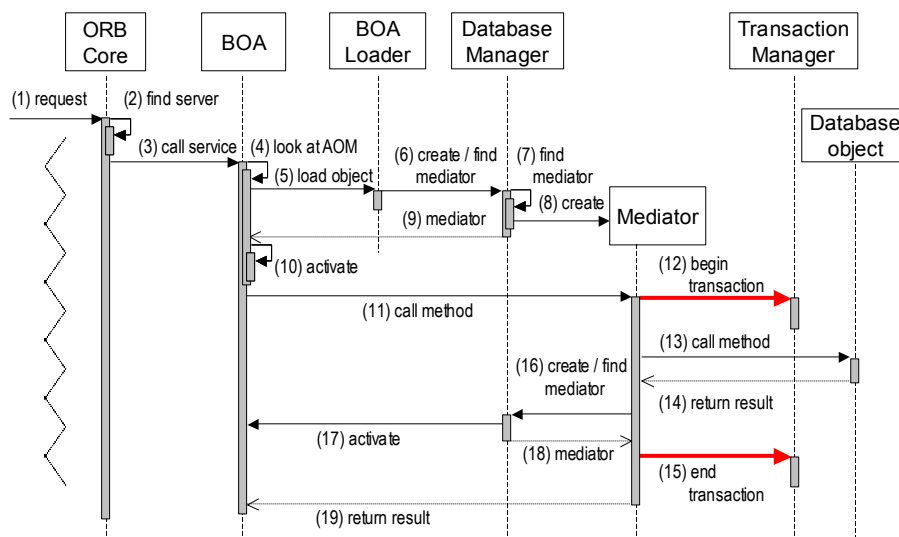


Figure 46. Demarcation of transaction boundaries during the request processing

XDA's transaction manager works in tight cooperation with ObjectStore's transaction manager. Since the latter works only for different UNIX processes, the XDA requires that every CORBA client have its own server. This correspondence is controlled automatically by the ORB daemon, registering the server in the *per-client-pid* mode.

As we can simply see from these rules, the transaction manager does not guarantee usual ACID features. This is not possible in this mode, since all transactions are processed automatically. The goal of this mode is only to guarantee that access to persistent data will take place and that it will be done always within a correct transaction. Actually, this mode is designed for „pure“ CORBA clients that do not want to care about transactions. Since the majority of transactional processing that most users will likely be doing with the XDA will be done in automatic mode, it represents an important part of the XDA's functionality.

5.4.4 Performance Improvements

As it was mentioned earlier, various design patterns can be used on the level of mediator objects' design to boost the CORBA/ODBMS system's performance. Further improving performance of the CORBA/ODBMS system, we have focused our attention on issues of transaction management, which proved to be one of the most crucial components of CORBA/database integrated system (see the results of measurements in Section 6.4.5). In our study, we tried to find and to implement additional means improving the default strategies for transaction management offered by ready products like ODAF or OOSA (see Section 3).

- *Transaction type management.* An approach adopted by ODAF does not differentiate the read-only and update transactions. Thus, every transaction started by the adapter is supposed to be an update transaction³². Such inflexibility has a negative influence on the performance and concurrency of applications where a considerable part of method invocations is *read-only*. As most of DBMS products support at least two kinds of transactions – *read-only* and *update*, it was decided to introduce the same separation to the adapter framework. For that purpose, some additional elements were introduced to transaction manager component. In particular, a special enumerated type indicating the type of transaction was introduced. Before starting to work with persistent data mediators, pass the transaction type information to the transaction manager, which then determines the kind of transaction (if any) to be open.
- *Persistent objects filtration.* In OOSA (as well as in accordance with ODAF default methodology) every method invocation on the CORBA server, which is linked with the database, causes transaction start, even if the destination object is not persistent neither deals with any other persistent object. In the environment where some considerable parts of CORBA server objects are not persistent this can have a substantial performance overhead as well as a worse concurrent access of multiple clients. The first idea was to make application server to decide *automatically* whether to start a transaction or not. However, this assumes that it can distinguish between persistent and transient objects, since the last objects do not need a transaction.

Request processing with use of introduced innovations related to persistence checking and transaction type management is illustrated below in Figure 47.

5.4.4.1 Transaction Type Management

Due to the approach adopted by ODAF the *read-only* and *update* transactions are not differentiated. Thus, every transaction started by the adapter is supposed to be an update transaction. As ObjectStore supports two kinds of transactions – *read-only* and *update*, it was decided to introduce the same differentiation to the adapter.

³² In OOSA things are not much better. The programmer can set the type (read-only or update) of transactions during the initialisation of the adapter. The type cannot be changed later dynamically and will be used for *all* transactions to be started by the adapter.

However, the transaction manager itself cannot automatically take the decision concerning the type of transaction to be open for particular method's invocation. Indeed, whether a method needs an update or read-only transaction cannot be deduced from its interface. Therefore, this decision is delegated to the mediators. Before starting to work with persistent data, mediators pass the transaction type information to the transaction manager, which then determines the kind of transaction (if any) to be open (Figure 47). A passed to the transaction manager value of the enumerated type indicates the necessary type of the transaction.

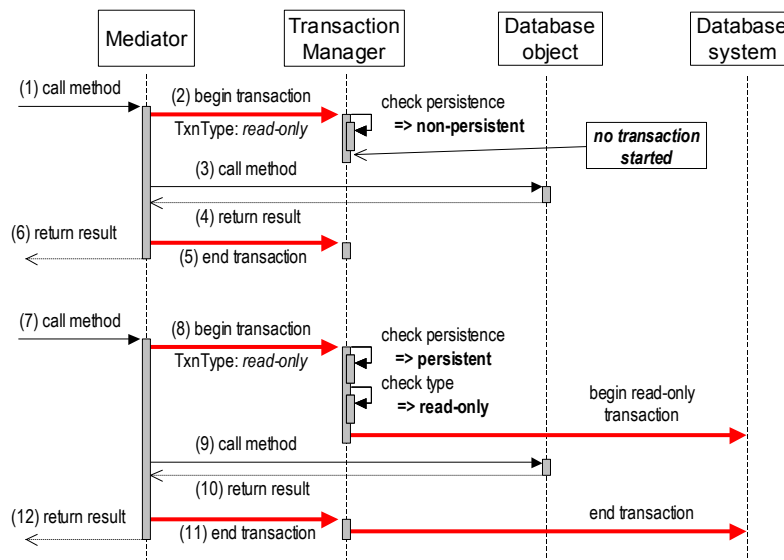


Figure 47. Persistence checking and transaction type setting mechanisms

5.4.4.2 Persistent Objects Filtration

The basic idea here is to make a transaction manager that will be able to control transaction boundaries automatically. To make such a decision it should also be able to make the distinction between persistent and transient database objects. This approach, however, is misleading since the object being itself transient can deal with other objects, which in turn can be persistent. In fact, the decision whether a transaction must be started is difficult to make without having application-specific information. Instead, some simple and effective means should have been provided to application server programmer to instruct the transaction manager whether it should begin transaction for any particular request of method invocation.

As a result, the same strategy as in transaction type checking was adopted. Since persistence is orthogonal to the classes and instances of the same class can be transient and persistent, a private flag has been introduced on mediator implementation classes. The flag will be set automatically by the adapter during object's instantiation. Upon a request for beginning the transaction, the transaction manager checks this flag and decides to start a transaction or not. This persistence check capability can be switched on/off at any moment dynamically.

5.5 XDA's CORBA Interface

XDA's IDL interface supports CORBA clients with a full set of methods for the manipulation with ObjectStore databases and persistent objects. The interface consists of the following particular IDL interfaces.

`XDA_Adapter` – includes basic adapter functions for manipulation with the adapter and a set of functions for opening/creating of databases and starting transactions.

`XDA_Database`, `XDA_DbSegment`, `XDA_DbCluster` – allows manipulating with databases, segments and clusters.

`XDA_Object` – is a base class for all mediators and picks together the functions common for all persistent objects.

`XDA_Type` – provides methods to access meta-information about available in the database classes.

`XDA_Transaction` – the objects represent concrete ObjectStore transactions.

`XDA_Collection`, `XDA_Set`, `XDA_List`, `XDA_Cursor` – represent XDA's collections interface for manipulation with groups of persistent objects.

5.5.1 Adapter Interface

The basic interface of the XDA is `XDA_Adapter`. It becomes accessible to the client immediately after binding to the CORBA XDA's server. The binding is performed in the traditional Orbix way with the help of the static function `_bind()`. This function takes parameters that specify the name and the location of the server implementation. For example, a client can bind to an XDA's server W as follows.

```
XDA_Adapter_var aVar;
aVar = XDA_Adapter::_bind ("Adapter:W", "xenon.iai.uni-bonn.de");
```

To access an attribute or an operation associated with an object, the client calls the appropriate member function of the C++ proxy class `XDA_Adapter` generated from the IDL. The proxy redirects this C++ call across the network to the appropriate member function of the server implementation. For more details about binding see [ITOrbix97].

After binding to the `XDA_Adapter` object, the CORBA client receives methods for the manipulation with databases, types, transactions and diagnostic information. Examples for their usage are presented in the following sections.

5.5.2 Database Interface

For the manipulation with databases, segments and clusters the XDA provides a set of four interfaces: `XDA_Database`, `XDA_DbSegment`, `XDA_DbCluster`, `XDA_Object`. Particular interfaces in this set represent for a CORBA client corresponding ObjectStore objects. Reimplementing a large part of methods from original ObjectStore classes (`os_database`, `os_segment`, `os_cluster`) in the IDL, the mediators make the methods accessible for CORBA clients and, thus, allow them to control the management of persistent objects.

Figure 48 presents an example for the manipulation with databases and persistent objects through XDA. It is a part of the original ObjectStore example that was rewritten for XDA. In comments, you can see related original code. Using the adapter's IDL interface the CORBA client can create a new ObjectStore database (1), then start a new transaction (2) and create a new persistent object of the class „part“ (3).

Sometimes CORBA clients might need to locate certain persistent objects in the database. As it was mentioned earlier ODBMSs use *entry points* to refer particular objects in the persistent memory (see Section 2.1.2.5). The same philosophy is used in XDA's CORBA interface `XDA_Database`. Using a function `create_root()` the CORBA client can create a root for every database object (4) and then find the persistent object associated with this root in the database (5). The interface `XDA_Database` presents also some operations for maintaining this association, for example an operation for retrieving of objects according to the name, an

operation for assigning a new name to the object and an operation for deleting the existing association.

```

// connect to the server
XDA_Adapter_var xda = ...;

// create a database
// os_database *dbl = os_database::create(example_db_name);
(1) XDA_Database_var dbl = xda->create_database ("test.db", 0664, 1);

// open a transaction
// OS_BEGIN_TXN(tx1, 0, os_transaction::update)
(2) XDA_Transaction_var tr1 = xda->begin_transaction (XDA_Transaction::update);

// create a part
// a_part = new(dbl, part_type) part(111);
(3) Part_var a_part = Part::_narrow (dbl->create_object ("part"));
a_part->part_id (111);

// create a root for the object
// dbl->create_root("part_root")->set_value(a_part,part_type);
(4) dbl->create_root("part_root", a_part, 1);

// find root
(5) a_part = Part::_narrow (dbl->find_root ("part_root"));
...

```

Figure 48. Example of manipulation with databases

5.5.3 Object Interface

XDA_Object is one of the main interfaces provided by the XDA. It provides a common interface for all user-defined mediator objects. All operations of the XDA_Object interface could be separated on two main groups.

The first group puts together operations for object maintenance. They are the functions for destroying, evicting and creating of new objects. Operations for object destroying and evicting always operate on the carrier object. They are specific for the interface XDA_Object, since from the safety reasons they are presented only on this interface. It makes these operations different from the other operations for creation of new objects. They could be also found on other XDA interfaces, such as XDA_Database, XDA_DbSegment, XDA_DbCluster. The main difference between all of them is in the location where the persistent database object will be created – in the default segment of the database, in a particular segment, in a particular cluster or in the case of XDA_Object interface – somewhere as close as possible to the target object.

The second group provides operations getting some information about the target object. It could be, for example, a database or a segment where the target object locates. More details about this interface can be found further in Section 6.2.2.

5.5.4 Type Interface

XDA_Type interface represents the main interface of the XDA MOP. The interface provides operations returning meta-information about the target type. For example, the name of the class, the names of available attributes, etc. The detailed description of this interface can be found further in Section 6.2.2.

5.5.5 Transaction Interface

The `XDA_Adapter` and `XDA_Transaction` interfaces present an ODMG-styled interface that lets CORBA clients control the transaction boundaries, explicitly beginning, committing and aborting database transactions. This powerful facility for controlling the transactions requires careful usage from the remote CORBA clients.


An example of the client code that uses different transaction control methods is presented in Figure 49. In fact, by default transactions will be controlled by the XDA's transaction manager (1). In this case, the client should not care about transactions at all. The XDA's transaction manager guarantees that every operation will be performed within the right transaction. Chained transactions are used in this mode (see Section 4.4.2.5).


When a client starts a new flat transaction (2), the responsibility of the transaction management is delegated from the transaction manager to the CORBA clients³³. In turn, the choice of transaction boundaries exerts a serious impact to the overall performance and concurrency capacity of the system (see Section 6.4). Controlling transactions manually, the client can limit the size of the rollback using nested transactions (3). In this case the failure of one sub-transaction is limited to just that sub-transaction.


```

// C++
XDA_Adapter_var xda;
XDA_Transaction_var tr1, tr2, tr3;

... // connect to the adapter and open a database

(1)  op1 (); // default mode - implicit transaction
      op2 (); // management by the XDA

      // explicit transaction management by the client
(2)  tr1 = xda->begin_transaction (XDA_Transaction::update);
      op3 ();
      op4 ();
      tr1->commit_destroy ();

      // nested transactions
(3)  tr2 = xda->begin_transaction (XDA_Transaction::update);
      op5 ();
      tr3 = xda->begin_transaction (XDA_Transaction::read_only);
      op6 ();
      op7 ();
      tr3->abort_destroy (); // abort of the transaction
      op8 ();
      tr2->commit_destroy ();

```

read-only and update transactions

Figure 49. Example of the client code that uses different transaction control methods

5.5.6 Collections Interface

Instances of container classes (`os_Collection` class' family) are mapped to corresponding IDL interfaces or to special representations (see Section 4.3.2.13) depending on preferences of a programmer.

When the second approach is chosen, IDL sequences are typically used to define several representations for the collection's value. The programmer declares them on IDL before the declaration of the interface representing the persistent class, which refers to this collection in one of its methods/attributes. Since all operations under sequences are generated by the IDL

³³ Hence, the XDA transaction management routine must be turned off.

compiler, as a result, all collection-specific operations like iteration or addition/removal of elements are performed locally on the client side, where the elements of collection are transported. The following Figure 50 shows an example where two different mappings by value (2, 3) are used for an initial ObjectStore collection (1).

```

// C++
class gt_Point;
(1) class gt_PolyLine {
public:
    os_Set<gt_Point*>* getAllPoints ();
    ...
};

// IDL
// representation by value
typedef double GTA_PointRep[3];

// sequence of representations by value
typedef sequence <GTA_PointRep> GTA_PointRepSeq;

// representation by value (default)
(2) typedef GTA_PointRepSeq GTA_PolyLineRep;

// representation by references
(3) typedef sequence <GTA_Point> GTA_PolyLinePRep;

interface GTA_PolyLine {
    GTA_PolyLineRep getAllPoints_PolyLineRep ();
(4)    GTA_PolyLinePRep getAllPoints_PolyLinePRep ();
    ...
};

```

Figure 50. Example of the IDL mapping for collections using the value-based approach

If the first interface-based approach is chosen, the overall picture can be more complex. The following code taking the same input as in the example above will be generated (Figure 51). For every collection type the XDA defines a separate ODMG-styled IDL interface, for example, `XDA_Set` and `XDA_List`. The interfaces provide all operations that are presented on the original ObjectStore classes to the CORBA client (2).

```

// C++
class gt_Point;
class gt_PolyLine {
public:
(1)    os_Set<gt_Point*>* getAllPoints ();
    ...
};

// IDL
// representation by interface
(2) interface XDA_Set;

interface GTA_PolyLine {
(3)    XDA_Set getAllPoints_Set ();
    ...
};

```

Figure 51. Example of the IDL mapping for collections using the IDL interface approach

In general, the IDL interfaces presume the original ObjectStore philosophy of interaction with collections. They allow CORBA clients to perform operations like insertion/removal of

elements, iteration over the collection elements, querying the collection, testing an element containment in the collection and others. For example, a separate interface `XDA_Cursor` containing iterator-specific operations can be used for iterating through the unordered collections. Figure 52 presents an example of using `XDA_Cursor` for iterating through an unordered collection from the previous example. The original `ObjectStore` code is presented commented at the top of the example. First, a CORBA client retrieves a reference to the collection (1). Then a corresponding cursor should be initialised (2) and created (3). Iteration through the collection (4) will be made similarly to the original code.

```

// gt_Point *p;
// os_Cursor<gt_Point*> c(pLine->getAllPoints());
// for (p = c.first(); c.more(); p = c.next()) {
//     printf("point coordinates: %d %d %d\n", p->x(),p->y(),p->z());
// }

XDA_Database_ptr db;
GTA_Point p;
GTA_PolyLine pLine;
CORBA::Any param;
(1) XDA_Set_var set = pLine->getAllPoints_Set();

XDA_CursorInit_ptr curInit = new XDA_CursorInit ();
(2) curInit->coll = set;
    curInit->behavior = XDA_Cursor::f_unsafe|XDA_Cursor::f_order_by_adress;
    param <<= *curInit;

XDA_Cursor_var cur = \
(3)     XDA_Cursor::_narrow (db->create_param_object ("XDA_Cursor", param));
    delete curInit;

(4) for (p = GTA_Point::_narrow(cur->first()); \
        cur->is_more(); p = GTA_Point::_narrow(cur->next()) )
    {
        printf("point coordinates: %d %d %d\n", p->x(),p->y(),p->z());
    }

```

Figure 52. Using `XDA_Cursor` for iterating through an unordered collection

The objective of the other methods/exceptions defined in collection and iterator interfaces is obvious and can be deduced from their signatures and corresponding `ObjectStore` documentation [ODIcg01]. However, the query method of the collection interface needs further explanation. Its signature is determined by the equivalent method of the `ObjectStore` collection class. A query method call takes a query string, which expresses the condition that evaluates each element of the collection. In `ObjectStore` typically the query string consists of one or more member variables' comparisons connected by boolean operators (&& and ||). Since the CORBA client sees only IDL representations of attributes of persistent classes, the mediator should convert them to the original `ObjectStore` attributes to form a legal query string. Then `ObjectStore` parses the query string at runtime and, if the syntax is incorrect, raises an exception. Otherwise, a pointer to a collection containing only those elements that evaluate to nonzero the conditions expressed in the query string is returned.

Table 15 summarises all pros and cons for both representation strategies. The interface approach presents a natural way for manipulation with collections imitating the original `ObjectStore` method. The most considerable drawback of the interface approach is its weak performance. Frequent remote calls to lightweight operations like the iteration over the collection's elements may be very wasteful. At the same time, the opportunity to perform queries on the server side can be extremely important in certain types of applications. Thus, interface-based approach should be used *only* when CORBA client and server are collocated,

or when the query option and other high-level collection operations are to be intensively used. In all other cases, the use of custom value-based representations is much more advantageous.

Value-based representations	Interface-based representations
operates with local data	operates with remote objects
less complicated interface	more functionality, possibility of querying
good performance	bad performance on large collections
custom implementation	ready to use implementation

Table 15. Tradeoffs between different representations of collections

5.6 The Spatial Object Database Adapter (SODA)

The extensibility of the first XDA prototype was evaluated by the development of a CORBA wrapper prototype for the object-oriented 3D spatial database *GeoStore* [BBC97]. The wrapper was developed on top of the XDA and was called *Spatial Object Database Adapter* (SODA) [BBB+98, SC00] (Figure 53).

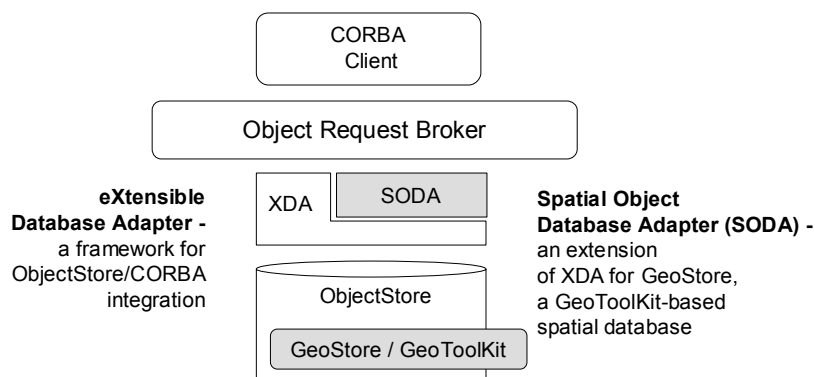


Figure 53. Architecture of the *Spatial Object Database Adapter* (SODA)

5.6.1 Implementing custom Managers

The XDA framework is an almost complete adapter for ObjectStore databases, providing wrapper programmers with a full set of necessary managers and pre-defined building blocks for developing new components. The core of the XDA consists of the main container module *XDA_Adapter* and four replaceable plug-in components – managers. For every manager the XDA provides a default implementation. It can be used in the most cases. However, if the target database has some specific requirements, for example to the creation of the databases, then the default manager should be replaced by the custom implementation. For this purpose all managers are designed as plug-in components that can be extended and replaced by the programmer.

Figure 54 presents a class diagram of the XDA’s extension for an object-oriented 3D spatial database *GeoStore* that is called as *Spatial Object Database Adapter* (SODA). Here all default XDA managers are replaced by custom SODA-specific managers. However, it is also possible to replace only particular managers, since they represent relatively independent parts of the XDA’s functionality. To guarantee compatibility to other XDA components the new defined SODA managers inherit abstract XDA manager interfaces.

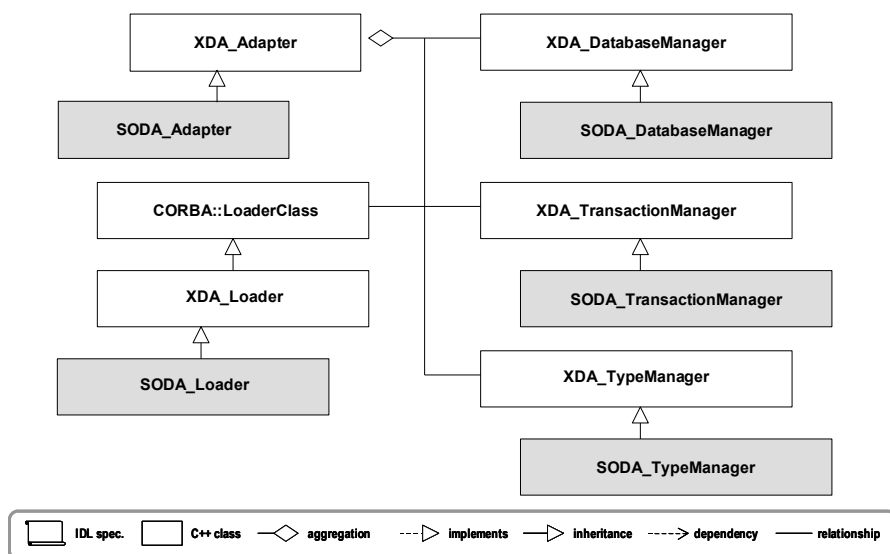


Figure 54. Class diagram of the *Spatial Object Database Adapter*

5.6.2 Implementing custom Mediators

Through corresponding pre-defined mediators the XDA makes original ObjectStore classes (e.g. `os_database`, `os_transaction`) accessible for a CORBA client. However, sometimes a database can have its own classes that encapsulate the original ObjectStore classes and extend their functionality. In this situation, the XDA allows a server programmer to redefine some parts of its native functionality. For example, *GeoStore* encapsulates an `os_database` class by a user-defined class to store an extended meta-information about a particular database. In the SODA prototype we defined our own class – `SODA_Database_i` that inherit and extend the functionality from the default XDA class `XDA_Database_i` (Figure 55).

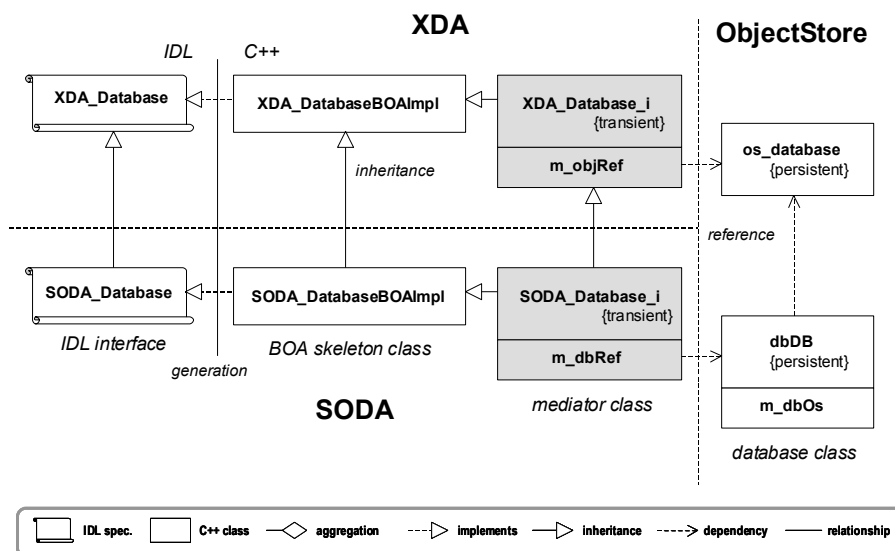


Figure 55. Extending the XDA with a new mediator class

5.6.3 Implementing abstract Mediators

If a proprietary scheme of the object management is necessary, then it is possible to extend the functionality of the interface `XDA_Object`. In SODA, it was done by defining a new interface `SODA_Object` that inherits the interface `XDA_Object` (Figure 56). New functions

need to be implemented in the corresponding implementation classes, for example SODA_Object_i and SODA_ObjectImpl.

Interface SODA_Object defines operations that are common and specific to all SODA objects. SODA_Object is an abstract class, which cannot have direct instances. Defining a new mediator template class SODA_ObjectImpl the server programmer keeps implementations of the mediator classes (for example, SODA_Point_i) free from the XDA-dependent details.

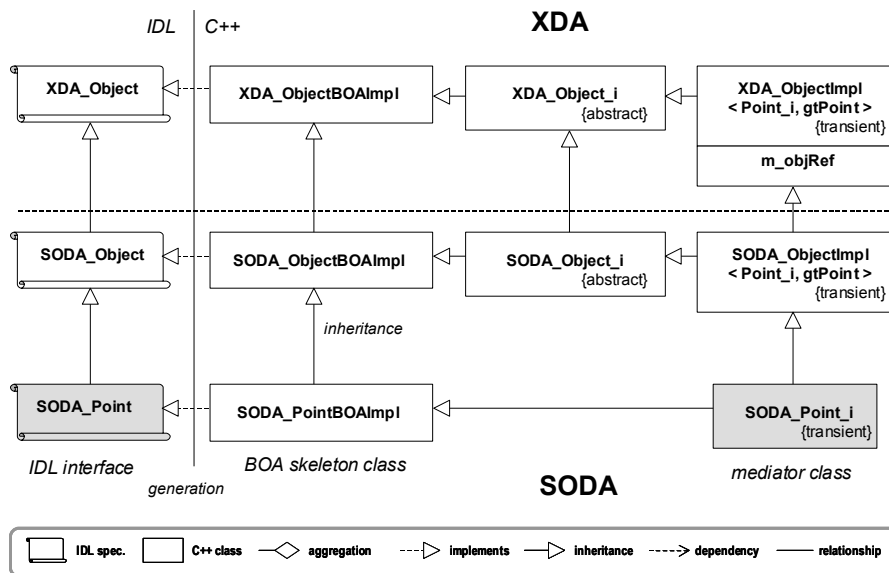


Figure 56. Extending the XDA with a new class for database entities

5.6.4 Issues with Multiple Inheritance and Downcasting

One somewhat complex situation can arise if one database class inherits from one or more other database classes. For example, DBStratum inherits from gtTriangleNet and some other classes (Figure 57) [SC00].

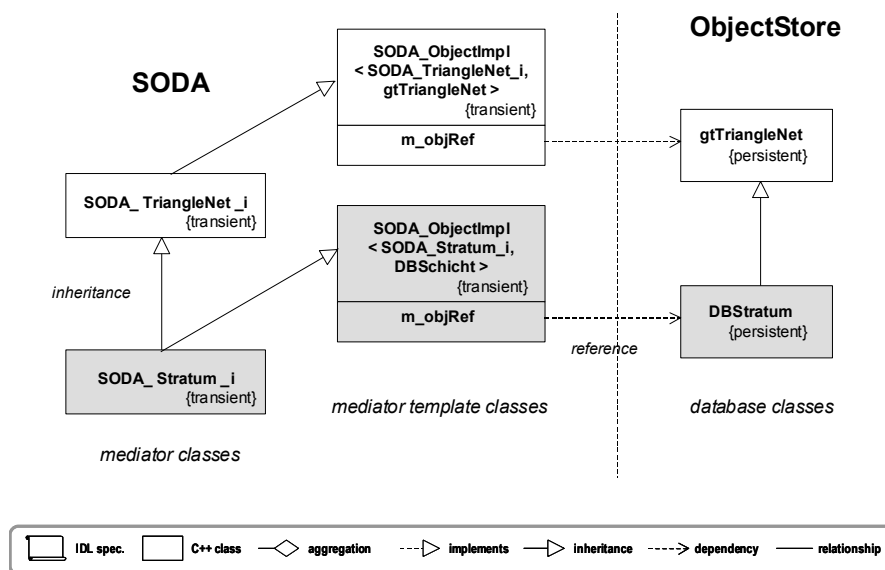


Figure 57. An example of multiple inheritance between database classes

Because of multiple inheritance a reference to an instance of `DBStratum` class will differ from the same reference cast to the base `gtTriangleNet` class. Mediator template classes will solve this situation using different references. The inheritance relation between database classes will be transferred to the inheritance relation between corresponding mediator classes. Therefore, in our example `SODA_Stratum_i` inherits from `SODA_TriangleNet_i`. In this case, the mediator for `DBStratum` will have two instances of mediator template classes that will work with the same database object through different references (`m_objRef`).

If a client has an IPOR reference to an instance of `SODA_Stratum_i`, then it can use this reference in corresponding operations that take an instance of `SODA_Stratum_i` as a parameter. The implementation of the mediator's operation receives a reference to an instance of a `SODA_Stratum_ptr` class that is not equal to an instance of `SODA_Stratum_i`. To help the mediator's programmer to downcast the incoming CORBA references, Orbix provides an abstract virtual function `_deref()` that is declared on all generated from IDL „proxy“ classes like `SODA_Stratum_ptr`. Therefore, overloading this function with functionality returning a reference to the `SODA_Stratum_i`, it is possible to implement a function downcasting `SODA_Stratum_ptr` to `SODA_Stratum_i`.

However, the both classes `SODA_TriangleNet_i` and `SODA_Stratum_i` inherit from the abstract `SODA_Object` class (Figure 56) and there is no pre-defined function in Orbix that could downcast `SODA_Stratum_ptr` to `SODA_Object_i` or `SODA_TriangleNet_i`. Therefore, the XDA defines the following schema of downcasting. First, the pre-defined function `_deref()` is overloaded only in `XDA_Object_i` and always returns a reference to this class. In turn, `XDA_Object_i` defines a set of abstract virtual functions `_abs_deref()`, `_imp_deref()` and `_os_deref()` that must be redefined in the child classes and return references to the top abstract class (`SODA_Object_i`), most deep implementation class (`SODA_Stratum_i`) and the associated ObjectStore class (`DBStratum`), respectively. Here it is assumed that if there is more than one level of abstract classes and downcasting to all these levels is necessary, then the wrapper's programmer can define its own downcast functions in the top abstract class. So, this schema allows to downcast incoming references to all levels of the inheritance hierarchy.

5.6.5 Evaluation of the BOA-based XDA Implementation

We took the BOA and tried to realise the XDA as an extension of this adapter providing the services essential for the manipulation with persistent objects. The outcome of this implementation is that the first version allowed us to identify the crucial problems appearing in the integration process. Four key issues identified to be crucial for the CORBA/ODBMS system functioning.

- **Generation, conversion and interpretation of object references.** To allow on-demand activation of dormant objects, the adapter must ensure that object references handed out to CORBA clients contain information on the location of the corresponding objects in persistent memory. Hence, the speed of the generation, conversion and interpretation of references to persistent objects that the adapter does is crucial for the performance of the CORBA/ODBMS system.
- **Object loading and activation issues.** The transience of mediator objects forces the adapter to take the responsibility for their management, i.e. activation/deactivation. Normally any CORBA object must be *activated* in object adapter before the first request for that object is delivered. The persistence of objects introduces essential changes to this scheme: object activation and object creation/location should be inherently separated. The constructor of a persistent object can be invoked only once when the object is created in a database. Once

created, the object can be subsequently loaded and activated/deactivated in object adapter as a usual CORBA object. Please see Section 2.2.8 for details.

- **Object deactivation issues.** The server-side CORBA objects that are not referenced by clients can (and sometimes should) be disposed in order to save the server's memory. Generally, the deactivation of objects seems to be more problematical than their construction. The program has clearly defined conditions when a given object must be constructed, but the question when that object should be removed from the server's memory remains open. Similar to the previous case, when the objects are persistent, they are deactivated rather than deleted. Indeed, the lifetime of persistent objects is not limited by the duration of the corresponding server process. In this situation, the mediating tier may either delegate object deactivation issue to the underlying DBMS, or introduce its own mechanisms.
- **Transaction management issues.** When a request for a persistent method arrives on the CORBA server, a transaction has to be open prior to the method's invocation (i.e. *before* the request arguments are unmarshalled) and subsequently committed after the results of method invocation are passed to the CORBA client (i.e. *after* the results are marshalled into the reply buffer). The naive approach of starting/committing a transaction from within the C++ member function that accesses persistent object does not work, because after the transaction's commit any persistent data returned by the operation may be removed from the memory *before* they will be marshalled into the CORBA reply buffer. The choice of the transaction boundaries has proved to be one of the most problematical issues critical for the performance of any CORBA/ODBMS system.

From the way those issues are handled depends the efficiency, flexibility and scalability of particular XDA-based integrated system. Further problems that were found in the first BOA-based XDA implementation can be listed as follows.

1. Limiting request dispatch capabilities of BOA
 - Activation = Instantiation
It is not possible to make an IOR without instantiation of a real servant
 - Duplication of AOM's information in XDA
It is not possible to implement a user-defined AOM or to use the existing BOA's AOM for controlling of mediators
2. 1:1 binding is efficient, but requires a lot of memory
 - How much mediators are necessary, 1:1 or 1:m ?
 - How to make the system scalable and capable to work with large numbers of persistent objects?
3. Intelligent transactions and resources control is necessary
 - How to determine the transaction boundaries?
 - Control for available resources (memory, persistent address space)
 - Control for concurrent requests from different clients
4. Manipulation with „a priori“ unknown persistent objects is necessary
 - Optimal marshalling and transmission for large data quantities

Code duplication. For each mediator class, the same code for servant creation and activation must be repeated, increasing the potential for errors.

Inflexibility: For each change in mediator class, the programmer must modify and recompile the server code, then stop and restart server processes.

Long dispatching time: Operations for creation, activation and deactivation of servants must be efficient and cooperate with the standard object adapter. There should be also no duplication of control information, such as tables of active objects.

High memory usage: An excessive amount of memory might be required to work simultaneously with large numbers of persistent objects.

Chapter 6

Improving the XDA Framework

Although the first version of the XDA was not an industrial-strength product, it allowed us to identify the crucial problems appearing in the integration process. Actually, the prototype allowed us to propose corresponding solutions, to develop and to evaluate new efficient methods for request dispatching and the management of transactions that are presented in this chapter.

6.1 Using the Portable Object Adapter

The second prototype of the XDA has been designed and implemented on top of the *Portable Object Adapter* (POA) – a new standard object adapter in CORBA. With this development, it was interesting to find how the new POA adapter can improve support for persistent objects comparing with the previous BOA-based version of the XDA.

6.1.1 Advantages of the Portable Object Adapter

Since the *Basic Object Adapter* (BOA) specification is too general to guarantee consistent implementations of persistent objects, each CORBA vendor filled in the details in proprietary way, making it hard to port server code from one CORBA product to another. An example of such an extension in Orbix is a Loader. The release 2.2 of the CORBA standard addresses these issues introducing a new object adapter – the *Portable Object Adapter* (POA) [OMGpoa97, SV9798]. The POA removes limitations of the old under-specified and uncompleted BOA and introduces many innovations. An introduction and basic concepts of the POA adapter were already presented in Section 2.2.6. Comparing POA with BOA here, we want to list only issues that are directly related with CORBA/ODBMS integration.

Server code portability. The BOA does not specify the form and the name of the skeleton base classes generated by the IDL compiler. Every CORBA vendor has developed proprietary naming rules, which make porting of server code between different CORBA products virtually impossible. The most important feature of the POA is standardisation of the naming of skeletons and other base classes. The IDL compiler of any POA-compliant CORBA product generates skeletons and stubs having unified naming conventions. The POA specification mandates that server-side names be formed by appending `POA_` on the beginning of the name of the related IDL interface, e.g. IDL interface `Foo` will be mapped to `POA_Foo` skeleton class.

CORBA objects and servants. The POA makes a difference between *CORBA objects*, which are *abstract* entities, and their implementations, which are written in a particular programming language and called *servants*³⁴. This clear separation allows a great degree of flexibility, since the lifetimes of CORBA objects and servant objects can be separated from each other: CORBA objects are *activated* and *deactivated*, whereas servants are *incarnated* and *etherealised*. The POA specifies an *Active Object Map*, which is maintained by an adapter and is used to map active CORBA objects to the associated servant objects.

³⁴ Previously referred to as *implementation objects*.

Object ID handling. The *Object ID*, being a part of CORBA reference, identifies the associated CORBA object within the scope of the adapter, which created the reference. The adapter uses the Object ID to associate client requests with target CORBA objects through the *Active Object Map* (AOM). In fact, the Object ID in POA characterizes the association between CORBA object and its servant. The POA may mandate one-to-one correspondence between CORBA objects and their implementations (servants). While this is a straightforward technique, it does not scale well in cases when a large number of CORBA objects have to be maintained by server³⁵. To address this, the POA additionally allows one servant object to implement multiple CORBA objects. In this case, the server does not have to maintain multiple servant objects in order to implement multiple CORBA objects. Instead, a single servant can examine the Object ID accompanying the incoming request, and switch its behaviour appropriately to the particular request.

Object activation mechanisms. The BOA lacks clearly defined object activation mechanisms, which resulted in providing proprietary solutions by some CORBA vendors. The POA specification eliminates that shortcoming by adding several alternative ways for object activation to be chosen by programmers. Besides providing mechanisms of explicit and implicit object activation similar to those supported by the BOA, the POA provides means for an on-demand object activation. To accomplish on-demand object activation, a POA registers a so-called servant manager, which handles all requests directed to objects, which are currently inactive. The servant manager is a CORBA pseudo-object that allows servants to incarnate the target object on-demand within the context of a request invocation and to return it to the POA. The programmer defines the behaviour of the servant manager and adapts it to specific application's requirements.

There are two kinds of servant managers: *Servant Activators* and *Servant Locators*. The functionality of Servant Activators is similar to Orbix loaders (see Section 2.2.8). After activation made by the Servant Activator each incarnated servant is associated with a CORBA object and is introduced to the AOM. The Servant Locator uses different semantics: the objects activated through the Servant Locator are active only for duration of the request processing. Once the request completes, the association between the CORBA object and the servant is destroyed (i.e. object remains inactive). The Servant Locator is invoked for every request to an object. Thus, in contrast to the Servant Activator, the Servant Locator takes over the functionality of the AOM, controlling the allocation of servants to CORBA objects.

Transient and persistent objects. The POA introduces the differentiation between transient and persistent objects. The lifetime of persistent objects is independent of the lifetime of the process in which it was created. It should be emphasised, that the „persistence“ here concerns only object's references, and does not imply that objects have persistent states. In fact, the POA reuses the notion of persistent references known since the first release of CORBA standard was published. Additionally, the POA specification defines transient objects, i.e. objects whose lifetime is bounded to the lifetime of the server process. Since the POA does not have to keep the track of activation information for transient objects, they involve less overhead than persistent objects, and still are useful in some cases (for example, call-back objects).

Summarising, the main reason behind POA being superior to BOA is that it allows programmers to build a consistent service for a servant's management that will be portable

³⁵ Today this is a commonly used technique, since the most C++ server applications use for each CORBA object a separate servant.

between different ORB products. However, none of the both adapters is designed to deal with such typical for databases features as segments and transactions. When we have to deal with persistent objects, we should always care about two very important aspects of their persistence. It is how the objects will be saved in the database (clustering) and how they will be accessed (transactions). Unfortunately, the both standard adapters either old BOA or new POA do not provide any support for these features. Therefore, the need for an extension of the standard adapter that will care about the object’s persistence is still necessary.

6.1.2 Architecture of the POA-based XDA

The second prototype of the XDA was made as an extension to the standard *Portable Object Adapter (POA)* (see Section 2.2.6). *ORBacus* [ITorbacus01] was used as the a POA-capable CORBA implementation [OMGorb01].

In general, POA replaces BOA and takes its place in the integrated CORBA/ODBMS system. A class diagram of the POA-based XDA is presented in Figure 58 below. The architecture of the new adapter is developed according to the main design principles presented in Section 4.4.2. Similarly to the previous XDA version, the core of the adapter consists of the main container module *XDA_Adapter* and four replaceable plug-in components – managers: an *object manager*, a *database manager*, a *type manager* and a *transaction manager*.

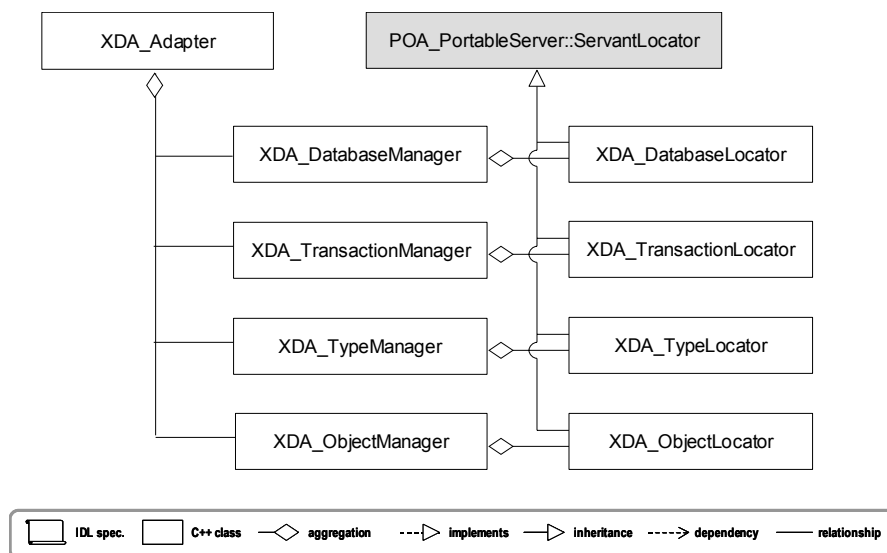


Figure 58. Class diagram of the POA-based XDA

The managers have the same names as in the previous BOA-based implementation, but based on cardinally different implementation concepts. For example, there are some differences in the design of the request dispatching. Whereas in the previous BOA-based version of the XDA the *loader* was responsible for the dispatching of the incoming requests to all types of objects, in the new POA-based version of the XDA this responsibility is separated between all managers. Now every manager has its own instance of the POA adapter equipped with a local object loader implemented as a specialisation of the POA’s *Servant Locator* (see Section 2.2.8).

6.2 New Techniques for the Implementation of Mediators

This section discusses new mediator design techniques that are intended to make integrated CORBA/ODBMS systems perform well and be more scalable, maintainable, and portable.

6.2.1 Implementing complex Static Mediators

Section 6.2.1.1 and the following sections describe an approach for isolating the implementations of the IDL interface from the XDA-related code to achieve better portability and separation of concerns.

6.2.1.1 Separating the Interface and the Implementation

C++ uses „class“ as the basis of subtype polymorphism and inheritance, but it has been pointed out that the overloading of class limits the expressiveness and makes its type system inflexible. Typically, a generated skeleton class will inherit from some common base class. Skeleton surrogates that interact with proxies in another address space must store the information needed to access them. It is tempting to put this data in the common base class object. However, generation of a class, which has some members either directly or inherited from a common base class, has two significant drawbacks.

The first drawback is that all object instances must carry this data even if they do not need it. This requirement could be a deterrent to using such an interface for high-volume objects, for example, for aggregation types such as class `os_collection` and class `os_set` presented on Figure 59.

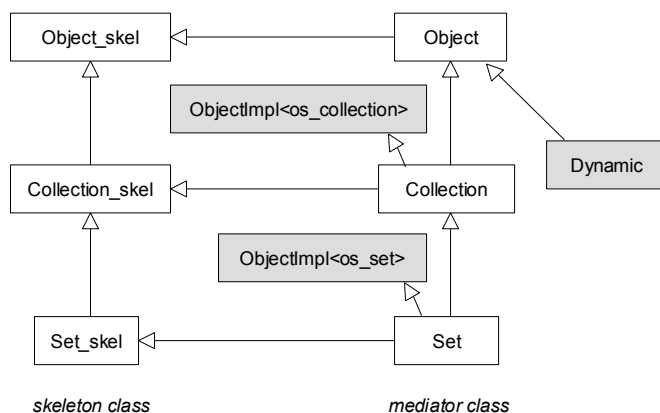


Figure 59. An inheritance hierarchy for classes implementing collection interfaces

These classes have some common public member functions that are typical for all database objects such as `type_of()`, `delete()`, etc. Thus, these operations may be gathered together into an abstract class `Object`, and derived from it. However, it is likely that the class `Collection` would inherit class `ObjectImpl<os_collection>` or something similar, and class `Set` would inherit class `ObjectImpl<os_set>`. Thus, the instances of the class `Set` must carry the part inherited from `ObjectImpl<os_collection>` even if they do not need it. A similar case occurs with instances of the class `Dynamic` implementing dynamic mediators. These objects are not accessible remotely through the skeletons, as they use DSI. Inheriting from `Object` we want to define a common class for manipulation with all server objects, but we do not want to burden them with unnecessary overhead.

The second drawback is that this schema requires virtual base class derivation, since in the case of multiple inheritance the derived class should contain only a single copy of the base class. Consequently, if multiple inheritance is necessary, then the usage of virtual inheritance is inevitable.

A strong separation of the inheritance hierarchy in two independent hierarchies for interfaces and for implementations should help us to solve the problem. This notion of complete separation of interface and implementation in C++ has been already explored in [Mart91] and further discussed in [Wald91, CHK96, CK00a and CK00b]. However, there seems to be no leading C++ model for separating interface lattice from implementation lattice.

Like others, our model also uses virtual base classes and multiple inheritance for separating interface and implementation. A delegation approach was used for binding mediators with corresponding database classes that do not allow any modifications. To illustrate the proposed model we took the previous example presented in Figure 59 and rebuild the inheritance hierarchy according to the new model. It is presented in Figure 60.

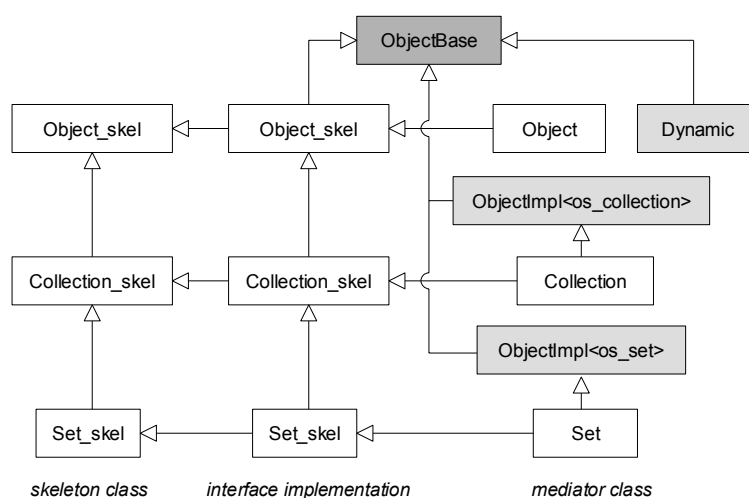


Figure 60. Separating inheritance hierarchies for interfaces and mediator classes

Basic differences to the previous schema are: the introduction of the new base abstract class `ObjectBase` and the separate hierarchy of classes implementing IDL interfaces. Since mediator classes inherit implementations of all IDL functions from the corresponding interface implementation classes, there is no more need for the inheritance between mediator classes itself. This design decision solves the first problem with duplication of unnecessary functionality in mediator classes. XDA-specific functionality from `ObjectImpl<...>` is made available in classes implementing IDL interfaces with the help of an abstract base class `ObjectBase`. Introduction of the abstract base class also solves the second problem with internal mediators. Inheriting from the class `ObjectBase` they are now free from unnecessary skeleton parts located in `Object_skel`.

The potential costs concern the usage of virtual base classes implementing IDL interfaces. Our own experience is that virtual base classes are expensive – at least in the C++ implementation that we use. An alternative to using multiple inheritance with virtual base classes is delegation. In our case, it not very useful because of increased number of objects. However, the new approach does not introduce new paths with virtual inheritance comparing to the old schema. Virtual base classes are not necessary even if the interfaces use multiple inheritance, so long as the base class contain no data.

6.2.1.2 A new Approach for the Implementation of Static Mediators

The basic idea of this schema is to separate implementations of IDL interfaces from implementations of mediators. The schema is presented in Figure 61.

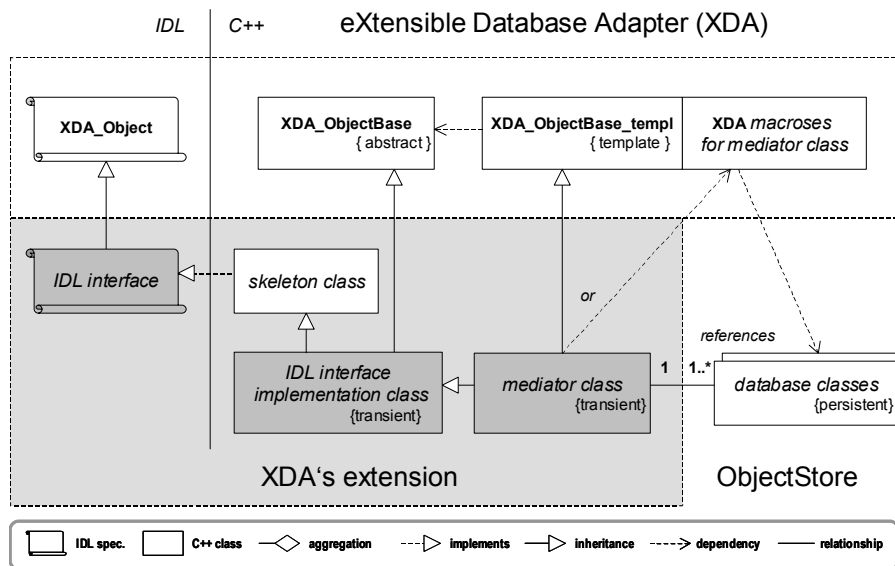


Figure 61. Implementation of the static mediators in the POA-based XDA

The functionality of the C++ class implementing a mediator, which was presented in Figure 38 earlier, is separated between two classes now. The first class implements the corresponding IDL interface and the second one implements the mediator. When the mediators are separated from implementations of IDL interfaces, it is possible to reduce the number of mediator classes and to assist extensibility by promoting the reuse of the implementation at the same time. XDA-specific functionality that is necessary for the implementation of the IDL interface (for example for resolving of persistent references) is made available through inheritance of the abstract class XDA_ObjectBase. Abstract functions declared on XDA_ObjectBase are implemented later in the mediator class that inherits their implementations from the XDA_ObjectBase_templ class or uses a set of equivalent macros.

Therefore, the mediator class in this schema plays the central role. It binds together the implementation of the IDL interface with all other XDA-specific functions that are necessary for the manipulation with persistent objects. Inheriting corresponding IDL interface implementation classes, the mediator classes inherit also XDA_ObjectBase. This inheritance allows the XDA managers to work with all mediators as with instances of this base class.

6.2.1.3 Multiple Inheritance and complex Class Hierarchies

Independence of IDL interface implementations from the mediator implementations makes a clean separation between their inheritance hierarchies and solves the problem of multiple inheritance. Indeed, a server programmer deals with four distinct class hierarchies: interfaces, their implementations, mediators and database classes. Two first interface-related hierarchies are based on subset relationships between IDL interfaces, which are viewed by all users including CORBA clients. The hierarchies for the mediator classes and corresponding database classes are based on code-reuse and are independent of each other and from the interface hierarchies. A principal schema of possible implementation of mediators for database classes, which use multiple inheritance, is presented in Figure 62.

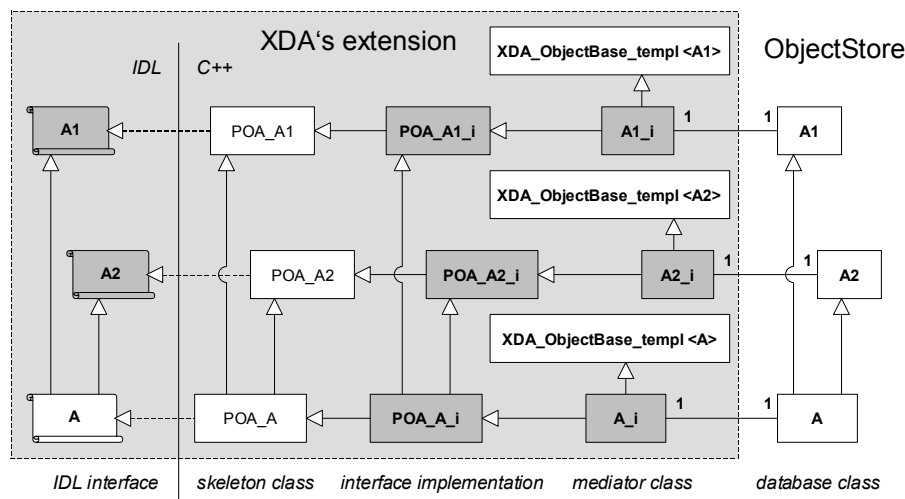


Figure 62. Implementing mediators for database classes with multiple inheritance

It is simple to see here that the relation of multiple inheritance is propagated from the database classes to the corresponding IDL interfaces, generated skeletons and their implementations. Corresponding mediator classes `A1_i`, `A2_i` and `A_i` do not keep this relation and therefore do not have a clash between implementations of XDA-specific functions. A diamond-shaped inheritance lattice, which occurs typically between sibling classes sharing information introducing a dependence on a unique common base class, can be also managed with the proposed schema. The difference occurs only in the case if abstract base classes are present. As opposed to an ordinary database class, an abstract virtual base database class can have only an IDL interface with its implementation. Interface `XDA_Object`, which is presented in Figure 63, gives an example of an abstract interface without a corresponding mediator.

6.2.1.4 Examples of Static Mediators

An example implementing static mediators for two classes from ObjectStore’s collection library is presented in Figure 63.

An inheritance hierarchy defined on database ObjectStore classes is directly mirrored onto a hierarchy of corresponding IDL interfaces (`XDA_Object`, `XDA_Collection`, `XDA_Set`) and C++ classes (`POA_XDA_Object_i`, `POA_XDA_Collection_i`, `POA_XDA_Set_i`) implementing functions defined in the IDL interfaces. The similar hierarchy will be also found on corresponding skeleton classes `POA_XDA_Object`, `POA_XDA_Collection`, `POA_XDA_Set`. Mediator classes (`XDA_Object_i`, `XDA_Collection_i`, and `XDA_Set_i`) bind implementations of IDL functions with XDA-specific functionality, thus building concrete classes for instantiation of the servants. Therefore, abstract IDL interfaces (such as `XDA_Object`) lack mediator classes since they cannot have real instances.

On the one hand, keeping this hierarchy on the IDL interface implementation classes seems to be quite natural providing the developer a clear concept for the implementation of IDL interfaces. On the other hand, this inheritance hierarchy is limited beyond these classes and does not follow forwards to the mediator classes `XDA_Collection_i` and `XDA_Set_i`. This independence of the inheritance hierarchies preserves the previously mentioned collision between their XDA-specific parts (`XDA_ObjectBase_template<...>`) for different database classes in the case of inheritance.

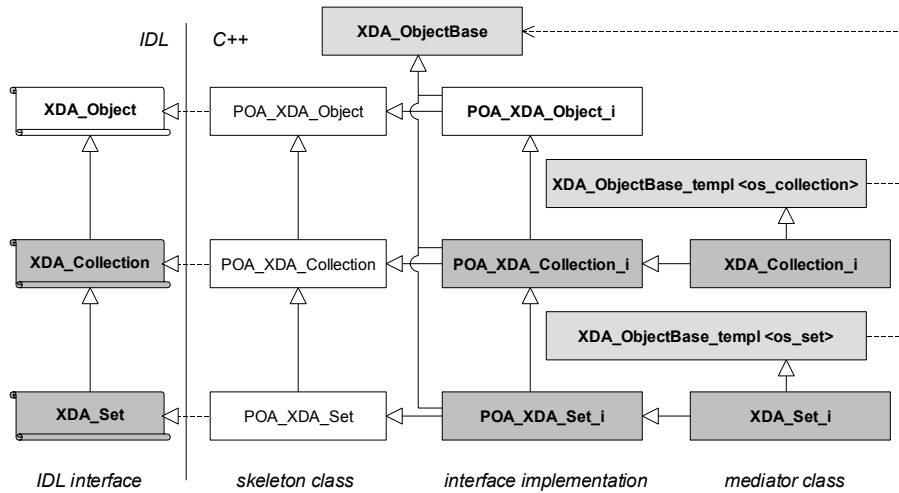


Figure 63. Implementing mediators for two classes from ObjectStore's collection library

6.2.1.5 Views: Multiple Interfaces and Composite Types

The example presented in Figure 63 gives us the first feeling of how mediators can be implemented, but there are more sophisticated bindings between IDL interfaces, mediator classes and database classes possible. In the contrast to the old BOA-based version of the XDA that supported only 1:1, 1:1:1 binding (Section 4.3.3.1), the new POA-based version of the XDA allows implementing of n:1:n bindings presented in Figure 64. In the following, we will discuss bindings of mediators to interfaces and to database classes separately named as *multiple interfaces* and *composite types*, respectively.

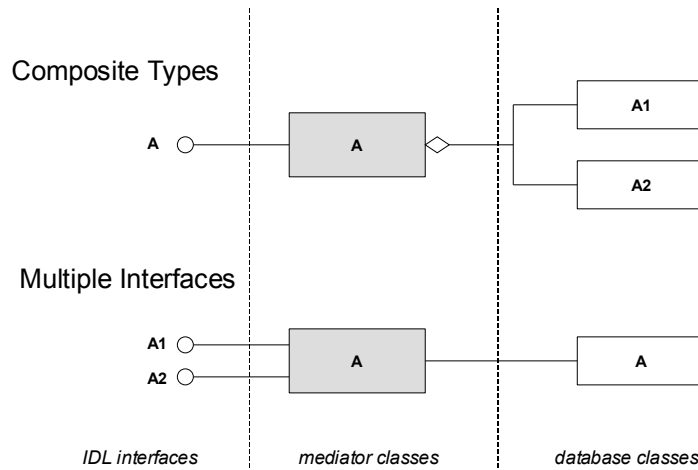


Figure 64. An example of possible correspondence between IDL interfaces, mediators and database classes

Multiple Interfaces

If one mediator class has I2, n:1:x binding to IDL interfaces, then we can say that the type realised by the given mediator class has *multiple interfaces*. Since mediator objects and interface components consume the operative memory at the server, their number is critical. Continuing with the example presented in Figure 63, let us try to simplify it and to reduce the number of mediator objects. Since `os_set` inherits from `os_collection` and IDL interface `XDA_Set` inherits from `XDA_Collection`, it is possible to use the instances of the `XDA_Set_i` as mediators working with both IDL interfaces. The modified schema is presented in Figure 65.

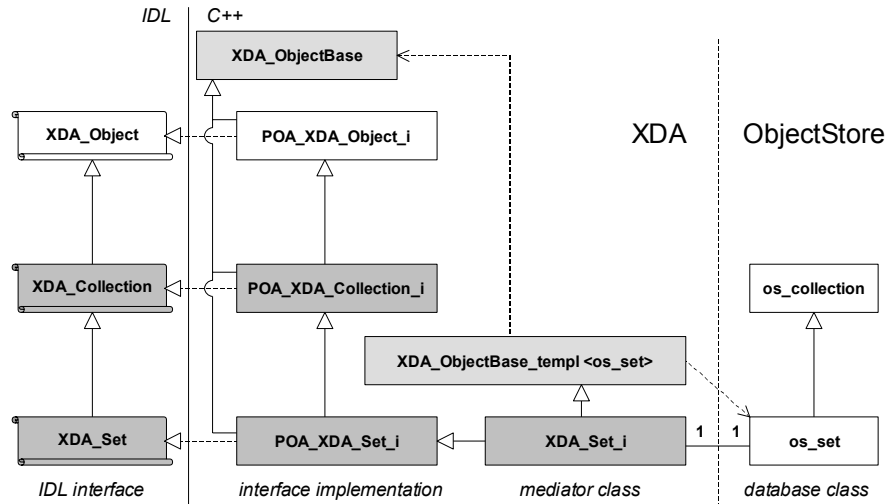


Figure 65. An example of implementing mediators for two IDL interfaces

According to the presented principle, a general schema for implementation of mediators supporting multiple interfaces is presented in the following Figure 66. Different from the previous example (Figure 65), in the general case a mediator can also support two independent interfaces that are not connected with each other by the inheritance. For example, Figure 66 shows a class *A_i* implementing mediators for interfaces *A1* and *A2*. In the schema the both interfaces inherit from the base interface *A*, but it is not always necessary. Therefore, it is possible to take an abstract interface *A* out from the schema promoting the inheritance relation from the interfaces *A1* and *A2* directly to the top base interface *XDA_Object*.

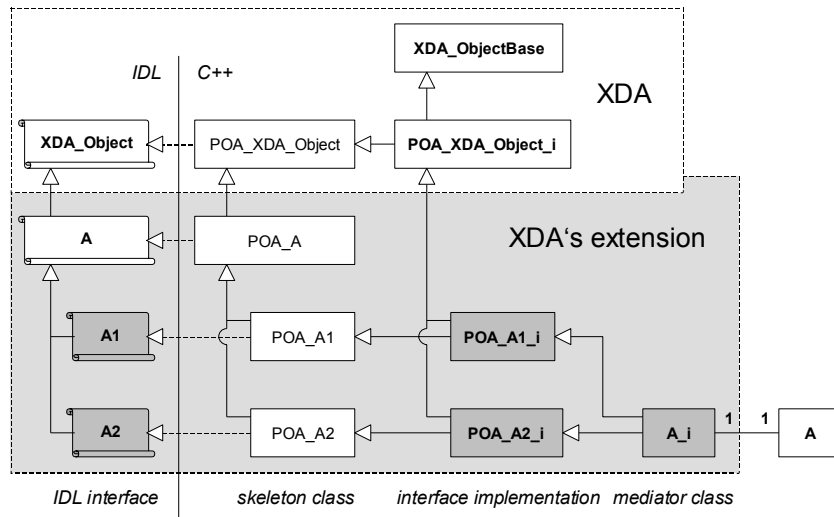


Figure 66. A general schema for implementing mediators supporting multiple interfaces

In the practice, the technique of multiple interfaces can be used, for example, for the implementation of mediators providing several IDL interfaces for different groups of users. In this case, the interface offered to the implementers can be larger than the interface offered to the users. Had this interface been for a realistically sized module in a real system, it would change more often than the interface seen by the users. Therefore, it is important that the users of a module be insulated from such changes.

The second case where the application for multiple interfaces can be useful is providing several IDL interfaces with different granularity. There are many cases where it is difficult to make a clean decision what kind of the interface is preferable: fine- or coarse-grained. Generally, from the reasons described above, objects in CORBA (mediator classes) tend to be much larger than they are at programming language level (database classes). In an extreme case, the whole database application could be a CORBA object. However, coarse object models are also not ideal and their drawbacks can force the programmer to select a finer granularity [HV99]. Using multiple interfaces it is possible to provide interfaces for both types. For example, applications that should work in both, remote and local environments can use interfaces with different granularity. This flexibility can be also used to support non-object-oriented applications, for example, relational or object-relational database management systems.

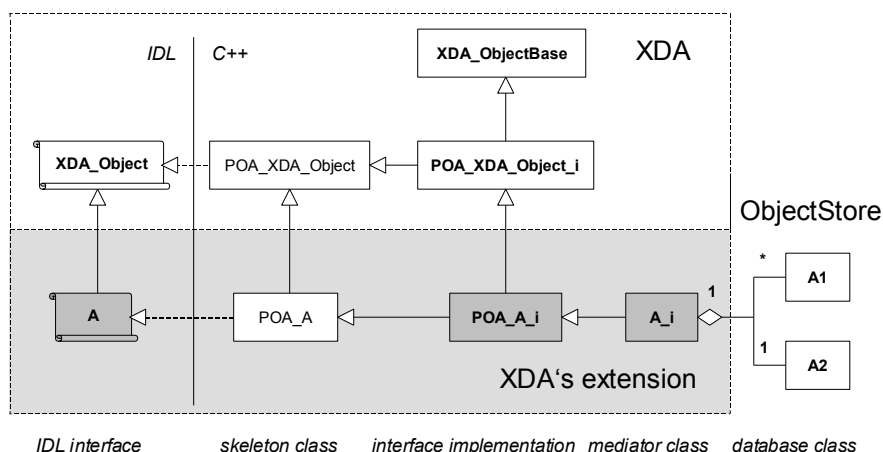


Figure 67. A general schema for implementing mediators for composite types

Composite Types

If one mediator class has I3, x:1:n binding to database classes, then we can say that the type is realised with the help of several database classes or is a *composite type*. Composite types represent the second way for reducing the number of mediator classes. A common schema for the implementation of mediators for composite types is presented in the following Figure 67. Comparing to the previous method (Figure 66), now mediators are implemented with the help of several database classes. Therefore, in this case one mediator object maintains more than one reference to database objects. For example, in Figure 67 a mediator class A_i will be implemented with the help of two database classes A1 and A2. Database classes here are arbitrary. It is also possible to use database classes having inheritance such as presented in Figure 65, but usually it makes no much sense, since in this case mediator objects will use two quite similar database objects. Hence, it is a very special case.

To illustrate the schema presented in Figure 67 an example from SODA adapter was taken. Figure 68 presents a class diagram for this example. Instances of class $SODA_Layer_i$ represent geological stratigraphic layers. The class is modelled in the database with the help of two different persistent classes $gtTriangleNet$ and $os_List<float>$. Instances of the class $gtTriangleNet$ represent geometry of the surface and instances of the class $os_List<float>$ represent density values bound with geometry of the given surface.

Actually, composite types can be seen as new database types composed through transient mediators. Practically it is useful for legacy databases where some additional type should be constructed, but at the same time keeping the existing database schema unchanged.

Combined with multiple interfaces in the practice composite types represent a powerful mechanism for the integration of existing databases with distributed environments.

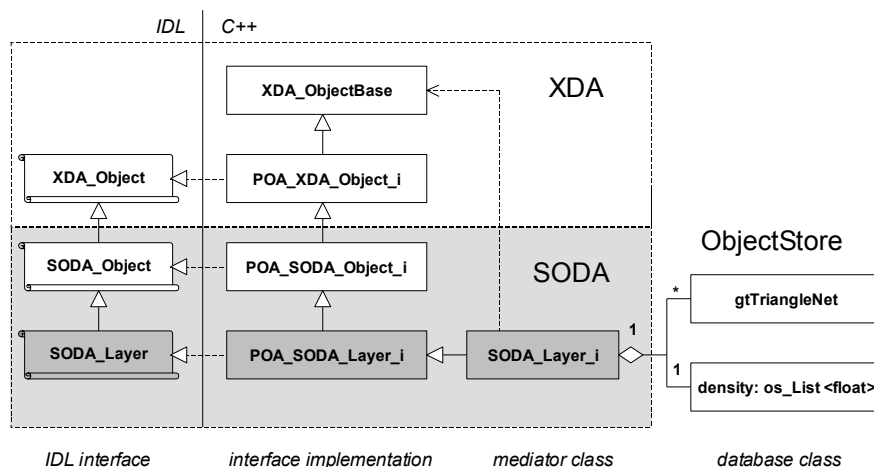


Figure 68. An example of implementing mediators for two composite types

6.2.2 Dynamic Mediators and Metaobject Protocol Interface

Static mediators provide an effective and natural way for remote interaction with database objects. With *static invocation*, clients know the IDL specification of the server at compile time and generate stubs that are necessary for the communication. However, if these specifications change, for example, by reflecting a schema's change in the database, then the both client and server parts must be recompiled and re-linked. Moreover, not all database classes have static mediators. Sometimes it is necessary to dynamically modify existing database classes, for example, add an attribute. Therefore, alternative mechanisms for dynamic runtime interaction without statically defined mediators were investigated.

Recent advances in fundamental software technologies, such as aspect-weaving software [AspJ03] and adaptive and reflective middleware, are beginning to address the problems outlined above. Adaptive middleware [KP94, PBJ98, ORS01] is software whose functional properties can be modified either:

- Statically**, for example to reduce footprint or to use and configure resources that can be optimised *a priori* (for interoperability with particular database objects); or

- Dynamically**, for example in response to changes in environmental conditions or requirements, such as changing component interconnection topologies; component failure or degradation; changing power levels; changing CPU demands; changing network bandwidth and latencies; and changing priority, security, and other needs.

The way in which usually such dynamic functional properties can be achieved is to introduce a *metaobject protocol* (MOP) [KRB91]. Metaobject protocols are interfaces to the system that give users the ability to incrementally modify the system's behaviour and implementation in runtime. This section provides a rough description of the XDA's MOP interface. More details about the interface can be found in [Tasc03].

6.2.2.1 Architecture of the XDA's MOP

XDA's *Metaobject Protocol* is an interface for dynamic runtime interaction with persistent objects of a priori unknown classes. The interface can be used for specifying and building requests, objects and classes at runtime, rather than using static stubs. The interface is built

with the help of static and dynamic communication mechanisms. Figure 69 presents the architecture of the XDA's MOP.

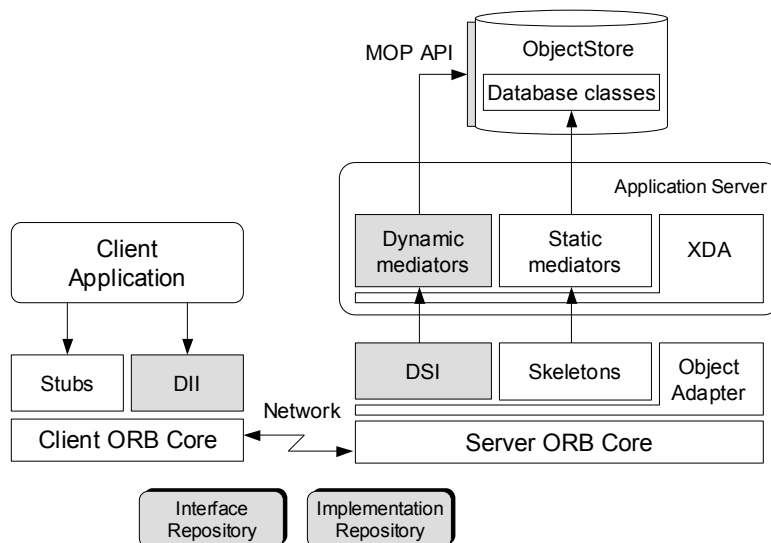


Figure 69. Architecture of the dynamic database interaction

Server-side Implementation

The server-side implementation consists of the following up-call interfaces, allowing calls from the ORB up to the database objects.

- The XDA static MOP interface (IDL skeletons)

This is a set of statically defined mediators providing the basic functionality of the MOP interface.
- The XDA dynamic mediator

It is a pre-defined mediator for dynamic runtime interaction with persistent objects of a priori unknown classes. The mediator receives all incoming requests through CORBA's Dynamic Skeleton Interface (DSI) and delegates them to the underlying database's MOP interface.

Client-side Interface

The client-side XDA MOP interface consists of the static IDL interfaces and supports the usage of CORBA's DII.

- The XDA static MOP interface (IDL stubs)

This is the XDA's static MOP interface, which includes the IDL interfaces for the dynamic manipulation with database objects, types, etc.
- The Dynamic Invocation Interface (DII)

DII can be used by the client for specifying and building a request at runtime, rather than calling linked-in stubs. The dynamic interface is necessary, if the object interface is unknown or does not exist in compile time.
- The ORB interface

The ORB interface allows functions of the ORB services like *Interface Repository* and *Implementation Repository* to be accessed directly by the client.

Possible Interaction Scenario

A possible interaction scenario between a client and a server can be imagined as follows.

1. First, using the ORB's Interface Repository or XDA's Metaobject Protocol (MOP) a client determines available database classes and their interfaces.
2. Then using Dynamic Invocation Interface (DII) a client can query at runtime the instances of these database classes that do not have statically defined mediators.
3. Using CORBA's Dynamic Skeleton Interface (DSI) the XDA processes such queries and forwards them to persistent objects with the help of ObjectStore's MOP.

6.2.2.2 Implementing Dynamic Mediators

The general schema for the implementation of static and dynamic mediators is presented in Figure 70. It shows the differences and similarities in the implementation of both types of mediators.

All mediators in the XDA inherit one common abstract class `XDA_ObjectBase`. This inheritance allows XDA's managers to work with all mediators as instances of `XDA_ObjectBase`. Thus, mediators of both types (static and dynamic) are processed by the XDA core in the same way.

Abstract functions (for example for resolving of persistent references) declared in `XDA_ObjectBase` are implemented later in concrete mediator classes specifically to their type. Static mediators use a set of macros. Dynamic mediators are represented by the instances of the class `XDA_ObjectDynamic` implement these functions direct in this class, since there are no any other dynamic mediators. All dynamic mediators in the XDA are instances of one class.

However, the cardinal difference between static and dynamic mediators is in how incoming requests are processed. In contrast to the static mediators, which use type-specific IDL based skeletons, the dynamic mediators do not have any compile time knowledge about the incoming requests and use for these purposes the *Dynamic Skeleton Interface (DSI)*. The DSI is a server side's analogue to the client side's DII, and provides a runtime binding mechanism for dynamic mediators to receive requests to the objects that do not have statically defined skeletons. Therefore, rather than using linked-in database functions, the dynamic mediators parse the DSI requests in runtime and delegate them to corresponding functions of ObjectStore's MOP.

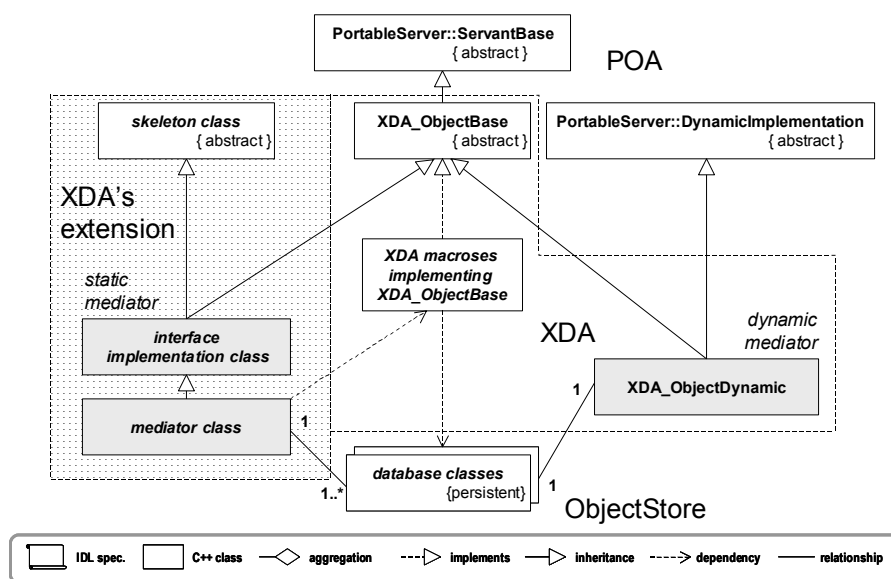


Figure 70. General schema for the implementation of static and dynamic mediators

6.2.2.3 XDA's MOP Interface

The XDA MOP interface is a set of statically defined IDL interfaces that allows to access database schema information dynamically.

XDA_Object

The interface `XDA_Object` provides operations for returning meta-information about the target object. For example, the operations, which return the type of the target object, and operations that allow dynamically to get/set an attribute of the object. Actually, it is possible to use these operations for all attributes, but their main purpose are attributes of the persistent object that are not presented in the IDL interface. Since these operations are implemented using ObjectStore's MOP, they are much slower than usual operations.

XDA_Type

The interface `XDA_Type` represents probably the main interface of the XDA' MOP. The interface provides operations returning meta-information about the target type. For example, the name of the class, the names of available attributes, etc.

XDA_Database

This interface provides operations allowing the client to access database schema information dynamically. For example, through these operations the client can retrieve information about existing types and create new types.

6.2.2.4 Advantages and Limitations of MOP

Reflection architecture of the MOP provides following *benefits*.

Changing a software system is easy. The MOP provides a safe and uniform mechanism for changing software. It hides all specific techniques such as the use of visitors, factories and strategies from the user. The interface supports not only read-access to the schema information, but also write-access, i.e. runtime evolution of the database schema.

No explicit modifications to the source code. It is not necessary to touch existing source code of the adapter when the database code was modified. Dynamic mediators provide a natural solution for dealing with schema evolution and for manipulating with a priori (at compile time) unknown types. This mechanism adds flexibility on the client side providing arbitrary queries.

Support for any kind of change. Using dynamic mediators, the server code could be made resistant against schema changes by the following procedure: when informed of a schema change, the adapter queries the data dictionary of the database to achieve the updated schema. Then, it dynamically constructs the new interfaces and announces them using the DSI. However, we believe that the generation of custom interfaces at runtime is only a theoretical possibility, because of the tremendous increase in the complexity of the adapter code.

However, reflection architecture has also some significant *disadvantages*.

Increased complexity of the client code. The dynamic invocation interface (DII) allows clients to construct and issue a request whose signature is possibly not known until runtime. This procedure violates the principles of objects transparency and increasing the complexity of the client code.

No operation calls are possible. The current version of the XDA' MOP does not allow to call an arbitrary database function at runtime. In fact, it is not a limitation of XDA's MOP, but the underlying ObjectStore's MOP.

Modifications at the meta level may cause damage. Even the safest looking changes to the schema information in multi-client environment may cause serious damage to the database and make other clients incompatible.

Lower efficiency. Reflective operations of the XDA's MOP are slower than equivalent non-reflective. This is caused by the complex relationship between the database and CORBA clients.

6.2.3 Generation of Mediators

In our approach, we use the CORBA level as a further level of abstraction by providing interfaces, which, in fact, are views onto the underlying database schema. Mediators defining such views and the mapping between interface and database schema must be adjusted after each schema change that is an error-prone operation. Therefore, to simplify such updates the methods for automatic generation of mediators are studied.

This section provides a rough description of the XDA Code Generator. More details and results of its evaluation can be found in [Bae03].

6.2.3.1 Architecture of the XDA Code Generator (CG)

The proposed approach assumes a semi-automatic generation of mediators, because in the general case the automatic generation of mediators is not possible. It is so, because the underlying database level usually does not provide all semantic information about existing database classes that are necessary for the generation of correct mediators.

For compatibility with the ODMG specification, the ODL language was selected as the basis language for describing the schema information. To provide necessary semantic information for generation of mediators, ODL was extended by necessary constructs. The extended language is called *eXtended Object Definition Language* (XODL).

In particular, it is the following constructs.

Transactions. The construct declares what kind of a transaction (read_only, update) is necessary for operations and attributes, since the last are represented by `get()`/`set()` operations.

Value Types. The construct declares how parameters or results of operations should be represented. For available variants, see Sections 4.3.2.10, 4.3.2.11, 4.3.2.12.

Restructuring Operators. The defined operators perform specific restructuring tasks defining the mapping between underlying database classes and XDA mediators (views). For details, see [Bae03, Roa00].

The usage scenario of the XDA Code Generator (CG) is illustrated in Figure 71. It shows an example of mediator generation for an existing database.

1. Using MOP of the underlying database management systems the XDA CG reads the database schema information and performs C⁺⁺ to XODL mapping. Then, the user has a possibility to edit generated XODL schema and manually set required additional constructs, such as transactions, value types and restructuring operators.
2. Using the XDA CG the user compiles the resulted XODL schema receiving all mediator components: IDL interfaces, Mediator header and rough implementation classes.
3. The IDL compiler for C⁺⁺ processes the IDL declarations to generate stub and skeleton codes. The programmer checks generated mediator implementations.

4. Finally, the XDA library and the server program are linked with the schema implementation. Thus, the change of the schema requires recompiling the XDA server and client.

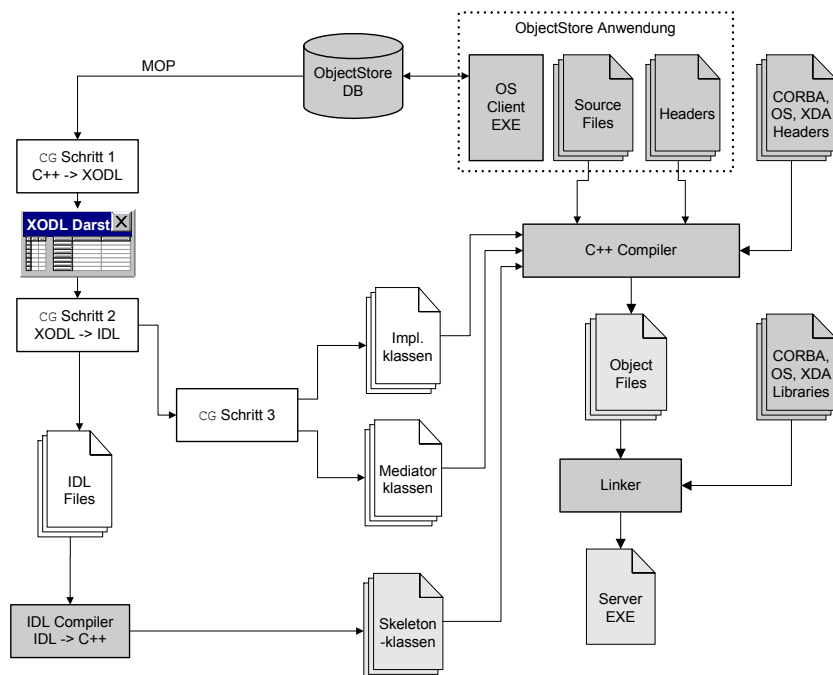


Figure 71. Building mediators with the help of the XDA Code Generator

The proposed method can be used also for the development of new databases. In this case, the following changes to the presented scenario are necessary: in step 1 can be omitted and the database schema will be designed directly in XODL language; after step 3 the ODBMS-specific preprocessor compiles generated C++ code and produces the schema implementation.

A simple test application was used to demonstrate the way the CG can be employed in practice to facilitate the ODBMS→CORBA integration [Bae03].

6.2.3.2 Summary

The primary purpose of the mediator objects is to adapt the interface of the classes used in pre-existing database application to the interface seen by CORBA clients. The *XDA Code Generator* (CG) generates mediator classes through which the ODBMS is integrated with the CORBA environment. Key elements of the CG are the specification of the XODL language, the definition of mapping rules and the principles of mediator class generation, which are closely interconnected.

Since IDL and ODL languages are not rich enough to keep all information about database classes that is necessary for the generation of mediator classes, an extension of ODL – *eXtended ODL* (XODL) was developed. The XODL language introduces new constructs specifying access type information to the attributes and necessary transactions. Defined C++ to XODL and XODL to IDL mapping rules allows generating of mediator classes and corresponding IDL interfaces.

Automatically generating mediator classes, their implementations and IDL interfaces, the CG radically simplifies the integration of existing ObjectStore-based databases with the CORBA environment. The generated classes take the form of static XDA mediators handled at

runtime by the XDA-made database adapter, which uses various transaction management techniques aimed to boost the integrated system's performance. Instead of tedious and error-prone hand coding, the programmers set their preferences, desired clients' perspectives, as well as implementation details concerning the mediator classes' layout as input information to the CG. The CG then generates ready-to-work, easily readable and editable IDL and C++ code significantly bursting the development process. The generated layouts of mediator classes can then be edited and customised by the programmer. Here the more complex is the input database application, the more tangibles are the benefits provided by the CG.

Another essential component of the CG's design is incorporation of XDA mechanisms providing additional flexibility and adaptability to the generation process. The CG allows programmers to manipulate the resulting system's structure by choosing various properties of the generated code, by defining CORBA clients' view on the structure of persistent objects, and by setting the general layout of the generated wrapper classes.

6.3 New Techniques for persistence-aware Request Dispatching

Comparing to the previous BOA-based XDA implementation, new techniques for request dispatching and mediator management were developed in the new POA-based version of the XDA. Using new facilities provided by the POA for extending the standard functionality, it was possible to realise more efficient techniques for request dispatching and mediator management.

6.3.1 Performance and Scalability Problems of common Techniques

When the XDA's object manager is asked by the native BOA runtime system to retrieve the missed mediator, usually it retrieves the necessary information from IOR and creates the mediator. At the end of every operation, after any results were marshalled into a reply message, the XDA issues release calls on all mediators instantiated while the current request was being serviced. Because the number of implementation objects in a database can be potentially very large, the XDA cannot keep in memory mediators to all the persistent objects it touches during the execution. Therefore, if the maximum number of mediators is reached, the XDA discards some of them. Whenever a discarded mediator is needed again, an equivalent to it will be instantiated by an object activation upcall.

There is a potential problem here. Assume an application server that runs for very long periods, possibly weeks or months, without ever shutting down. Using a servant manager, which creates mediators on demand, means that it adds each servant to its Active Object Map. Chances are that eventually some client or other will touch so many database objects that the next mediator does not fit in memory. The memory consumption for all mediators may be more than we can tolerate, so the server must be able not only to bring the mediators into memory on demand, but also to evict them if memory runs out or mediators have been idle for some time.

The Evictor pattern [ITev99] describes a general strategy for limiting memory consumption. The basic idea is that we use a servant manager to instantiate servants (mediators) on demand. However, instead of blindly instantiating a new servant every time it is called, the servant manager checks the number of instantiated servants. If the number of servants reaches a specified limit, the servant manager evicts an instantiated servant and then instantiates a new servant for the current request.

This technique we have implemented and evaluated in the first BOA-based version of the XDA. We have found that a longer evictor queue permits more mediators to be active in memory and results in better performance, whereas a shorter queue drops performance but also reduces the memory consumption. It must be aware, however, of one potential pitfall: if the evictor queue is too small, performance will fall off dramatically. This happens if there are more objects being used by clients on a regular basis than the server can hold in memory. In that case, most operation invocations from clients cause one mediator to be evicted and another mediator to be instantiated.

Since these operations of instantiation and discarding of mediators use dynamic memory, performance of this method suffers from typical problems occurring during frequent operations with dynamic memory [Alex01, May96, May98 and Sutt00]. There is a potential problem for scalability and performance of request processing. On the one hand, it is very expensive, since it is a pair of `delete()`/`new()` operations. It is possible to imagine a client populating a new database with several thousands of new objects. This means that although the application server initially starts up without any instantiated mediators, ongoing activity causes all mediators to be faulted into memory eventually. The memory consumption for all these objects may be more than we can tolerate, so the server does not scale and we will need to shut it down periodically to reclaim memory.

On the other hand, frequent operations for allocation/de-allocation of dynamic memory increase memory's fragmentation and follow to the problems with application's scalability. The problem is, the built-in operators `new()` and `delete()` are general and perform badly allocating small objects. In particular, the default allocator is notoriously slow and has a significant bookkeeping overhead for the management of allocated memory blocks. Usually it is about 4 to 32 extra bytes for each block. Therefore, if 1024-byte blocks are allocated, the per-block space overhead is insignificant (0.4% to 3%), but for 8-byte blocks it becomes 50% to 400%, a figure big enough to make the programmer worrying about the allocating of many such small objects.

Another possible way of achieving high performance without having problems with dynamic memory is to use the default servants supported by a POA with the `USE_DEFAULT_SERVANT` policy value. Default servants allow handling invocations for many different CORBA objects with a single servant (see Section 11.7.4 in HV99 for details). Default servants go a step beyond the servant managers in reducing memory requirements because they eliminate both the Active Object Map and the one-to-one mapping from object references to servants. The price of the default servant technique is that unless the server uses an aggressive threading strategy, object invocations are serialised on the default servant, so the invocation throughput will drop. However, default servants make it possible to create lightweight implementations that allow a server to scale to millions of objects while keeping memory consumption very low.

Since the both common techniques have their characteristic problems, a new technique based on the synthesis of strong sides of the both techniques is proposed. The basic idea is to use a fixed number of existing mediators for serving all requests avoiding operations with dynamic memory. Separating all mediators into independent groups according to their type, we are capable to reuse them serving several request. Since the mediators have no state besides references, it is possible to switch them to other database objects of the same type. Maintaining such a pool of type-oriented groups of mediators is essentially just another way to keep an upper bound on the number of mediators in memory at any given point in time.

To implement this approach, we must be able to replace the standard mechanism for request dispatching by a new that will be oriented to work with persistent objects. Using POA's *servant managers* it is possible to make a custom map for active objects and completely replace

the POA's default mechanism. Implementing an own pool of mediators with our own request dispatching oriented to persistent objects it is possible to speed up request processing. For realisation of the proposed mechanism, we should use the following POA features.

- An application server can have several POA adapters, each for a certain kind of mediators.
- Using POA's *servant managers* the programmer can define the user-defined mapping of CORBA objects to servants objects.
- The creation of CORBA IORs (or CORBA objects) is separated from the instantiation of servants. It is possible to create a reference avoiding the creation of a real servant object.

6.3.2 Using multiple POAs for more efficient Request Dispatching

The first difference between the old BOA and POA adapters that can help us to improve performance of request processing is to have several POA adapters, each for a certain kind of mediators.

The rate of requests that a server can handle is an important aspect for distributed client-server applications. Since the old BOA adapter could process incoming requests sequentially only one by one, POA tries to solve this problem introducing a new schema of request processing allowing to have several POAs inside of one server process. Thus, a single application may actually contain several POA instances organised into a hierarchy of child POAs descending from the Root POA that will be maintained by one or more *POAManagers*. As it was shown early in Figure 12, each POA has an associated with its *POAManager* that essentially acts as a faucet or valve allowing the programmer to control the flow of requests into the POA.

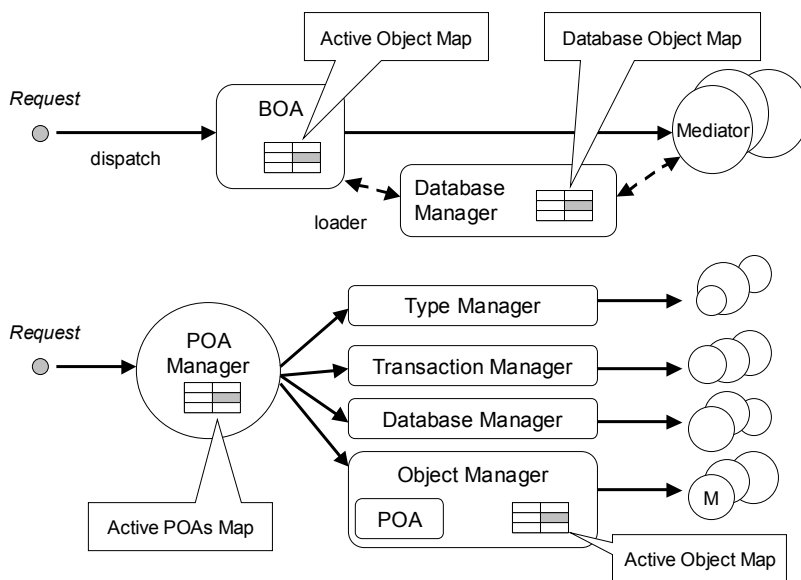


Figure 72. Using several POA adapters for more efficient request dispatching

The main idea is to use this POA's facility to improve performance of request dispatching by separating the requests to objects of different types at the first stage. The structure of XDA's managers already provides a required separation between different types of mediators. It is necessary only to equip every manager with an own POA. Figure 72 presents the idea

graphically comparing the proposed schema for request dispatching with the old schema supported by the BOA-based version of the XDA.

Format of Interoperable Persistent Object Reference

To support the new schema for request dispatching some changes in format of *Interoperable Persistent Object References (IPOR)* were necessary. It is presented in Figure 73. One of the components specified by CORBA encodes the POA, which created the given reference. For all incoming requests the *POA Manager* automatically extracts this *POA ID* from the object reference and uses it as a key in his table of active POAs to find a corresponding POA's instance (Figure 72). Comparing to the previous BOA-based version (Figure 39) some additional parameters are added. For efficient request dispatching *Class ID* was extended by an integer *Index* that is used to find the class of the object. *Object Marker* is represented by a sequence of *Persistent Object IDs* to the corresponding database objects. Details about the using of these parameters are presented later in this chapter.

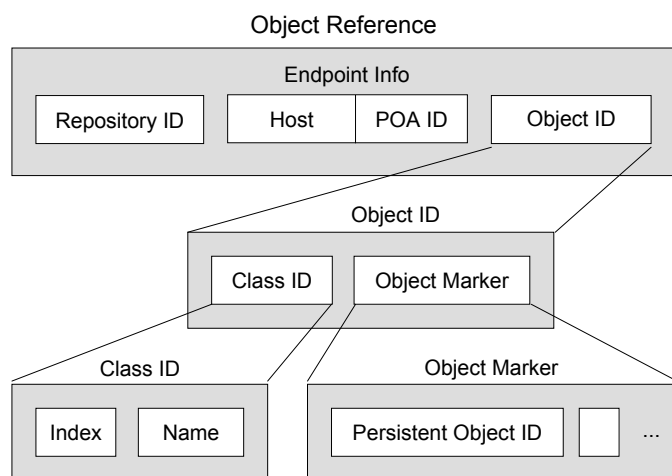


Figure 73. Format of the Interoperable Persistent Object Reference (IPOR)

6.3.3 Using POA Servant Managers for Request Dispatching

Separating incoming requests according to the possible type of objects the *POA Manager* reduces the search in the following steps performed by the selected manager. Thanks to this „pre-selection“, the manager can be highly specialised on the management of objects of the given type. For example, the *Transaction Manager* can expect that it becomes a reference to an object representing a database transaction. Using the second advantage of POA adapters the programmer can define the user-defined mapping of CORBA objects to servant objects.

A POA that has the `USE_SERVANT_MANAGER` policy value allows creating of custom components – called *servant managers*, which actively participate in the process of determining object-to-servant associations (for more details see HV99). A servant manager is a user-defined local callback object that the application registers with the POA to assist or even to replace the function of the POA's own *Active Object Map (AOM)*. When a POA attempts to determine the servant associated with a given target object, it calls back to the application's servant manager to obtain the servant. Depending upon the POA's value for the `IdUniquenessPolicy`, the servant manager can either provide the POA with a newly created servant or reuse an existing one. Either way, it returns the servant as the result of the up-call, which the POA uses to complete the request invocation.

After the invocation completes, the POA either retains the association of the servant and the CORBA object in its AOM or throws the association away, meaning that the next invocation

on the object will again require the services of the servant manager. It depends on the type of the used servant manager. There are two basic types of managers – `ServantActivator` and `ServantLocator`. Therefore, the basic difference between them is that the first one works together with the POA’s own AOM and the second completely replaces it.

When a POA with the registered `ServantActivator` receives a request for a target object, it consults its AOM to see whether a servant is already available for that object. If none is found then the POA invokes the `ServantActivator` to find or create a suitable servant’s instance. Therefore, `ServantActivator` is similar to the *Loader* that we have experienced in BOA-based version of the XDA.

A POA with the registered `ServantLocator` does not store object-to-servant associations in its AOM, so it must invoke its `ServantLocator` for each incoming request. Evaluating both types of managers, we have found that for applications dealing with persistent objects complete replacement of AOM in the case of usage `ServantLocator` can significantly improve performance of the dispatching.

The servant location mechanism in the XDA was realised using `ServantLocators`. Since the implementation schema is roughly the same for all managers, let us discuss the realisation of the manager on the example of the *Object Manager*³⁶. The UML class diagram for *Object Manager* is presented in Figure 74.

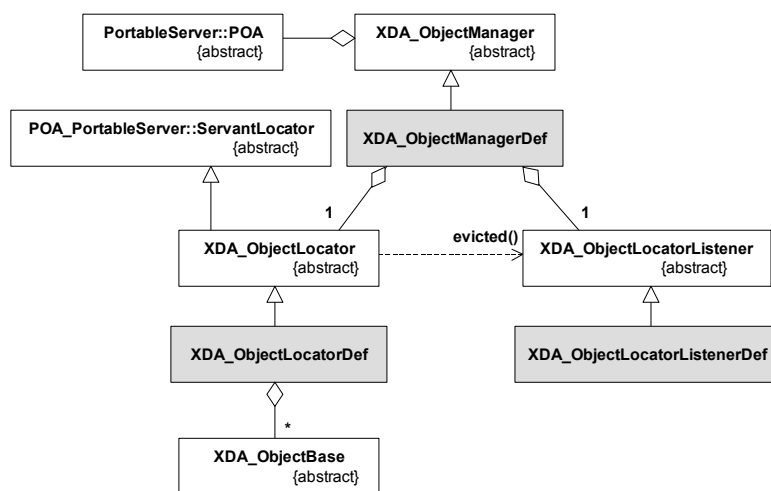


Figure 74. Class diagram for the XDA ObjectManager

The manager is implemented using an abstract class `XDA_ObjectManager` that defines an interface and its default implementation realised in the class `XDA_ObjectManagerDef`. Defining an abstract base class, we follow here the concept of the framework presented in Section 4.4.2. The concept of replaceable plug-in components spreads further to the default realisation of managers itself. For example, implementing a new class inheriting from the class `XDA_ObjectManagerDef`, the programmer can replace the default realisation of *Object Manager* or only parts of it.

As it is shown in the diagram, default implementation of every manager has two replaceable components – *Locator Listener* and *Locator*. They are represented by the instances of the classes `XDA_ObjectLocatorListenerDef` and `XDA_ObjectLocatorDef`, respectively. Realising the interface defined in the class `POA_PortableServer::ServantLocator` the *Locator* represents a custom servant manager. It will be invoked by the POA every time when an

³⁶ Actual request dispatching algorithms that the managers use depend on the type of managed objects and, therefore, can be different from manager to manager.

incoming request comes. The *Locator Listener* defines an interface for a component that is able to react to the operations performed by the *Locator*. This component is replaceable by the programmer. If the *Locator* decides to evict a mediator, it notifies the *Locator Listener*, which in its turn, for example, can prevent eviction of certain mediators.

The typical control flow diagram for dispatching of requests to user-defined database objects in the POA-based XDA is presented in Figure 75.

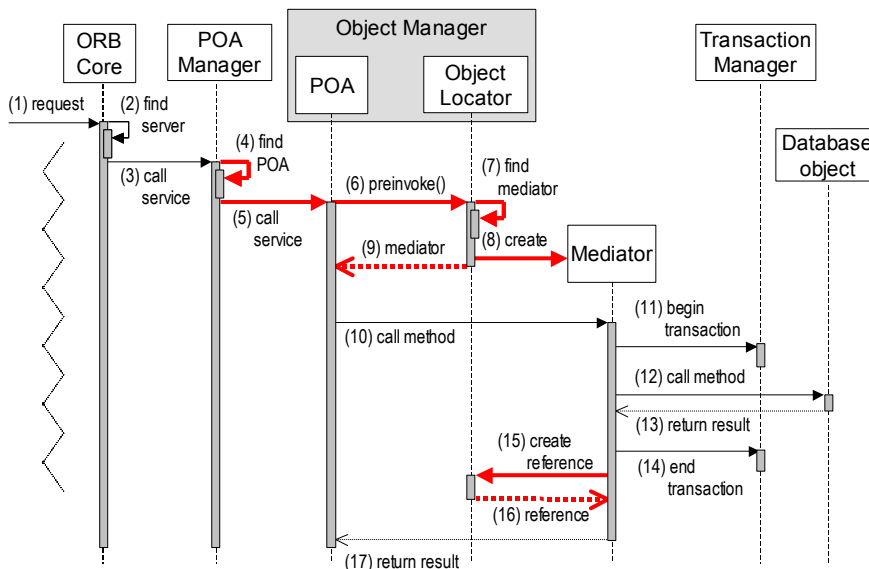


Figure 75. Control flow diagram of request dispatching for user-defined database objects in the POA-based XDA

Using *ServantLocators*, actually, we have replaced the default POA’s AOM by our own implementation. Now client invocations arriving for the various database objects are processed by the corresponding servants manager – *Object Locator*, which uses our own mechanism for request dispatching. The corresponding to the manager POA does not maintain an AOM. Instead, the POA invokes the *preinvoke()* and *postinvoke()* operations on the servant locator on every request. The job of *preinvoke()* is to return a pointer to the servant that should handle the request, whereas *postinvoke()* has the job of cleaning up after the operation completes. In our implementation, *preinvoke()* does all the work and *postinvoke()* is empty.

Comparing this schema with the previous schema for request dispatching in the BOA-based XDA presented in Figure 40 it is simple to find the differences between them. The first important difference is that the operations for the location of objects are now concentrated in one place, in *Object Locator*. There is no more duplication of information about active mediators. Now there is only one map of active mediators, which is tuned-up to the persistent objects and located in the *Object Locator*.

The second difference is that the creation of CORBA IORs (or CORBA objects) is separated from the instantiation of servants. The *create_reference_with_id()* method is used on a POA with a policy that accepts a custom *Object ID* generated by the *Object Locator*. This feature allows to generate IPORs without creating real mediators and searching in the tables of active mediators as it was inevitable in the BOA-based version (operations 15 and 16 in Figure 75).

6.3.4 Persistence-aware Method for Request Dispatching

Since default POA's AOM are usually realised with the help of plain hash tables and use *Object IDs* as search keys, this strategy can be very inefficient for applications containing many thousands of objects. In contrast to the standard technique, using the structure of *Object IDs* and separating it to several keys, it is possible to organise the search more efficiently. Thus, the method for request dispatching, implemented in the XDA, reduces the search time using the three-like hierarchy using *POA ID*, *Class ID* and *Object Marker* parameters extracted from *Object ID* as search keys.

Figure 76 illustrates both methods. Since the number of possible mediator objects can be very large, one large AOM for all objects that is used in default case was separated to several AOMs organised in three-like structure. This approach improves the scalability of the dispatching. At the same time, its performance on persistent object references is better than in the standard case. In particular, assuming the usage of hash tables and estimating performance of the standard hash table as $O(c_s)$ in the following discussion it will be shown that performance of new approach be estimated as $O(c_n)$, where c_s and c_n are constants and $c_s > c_n$.

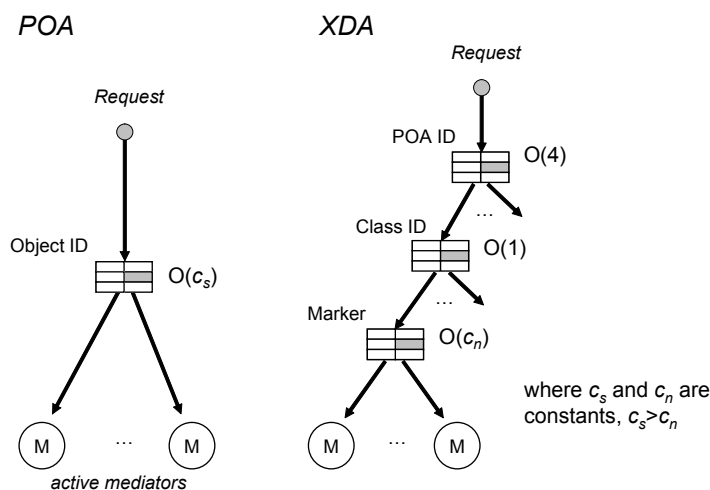


Figure 76. Comparison of the standard POA and XDA schemas of request dispatching for user-defined database objects

The implemented in the XDA method for request dispatching processes all incoming requests in three steps. Using *POA ID* at the first step all requests are dispatched between four XDA managers: *Database Manager*, *Transaction Manager*, *Type Manager* and *Object Manager* (Figure 72). The following steps will be performed by a manager selected in the first step and depend on the type of managed objects.

Figure 76 presents a general schema for dispatching the requests. All available mediators are organised in pools according to their type. We name them as *extents*. An *extent* can contain one or more active mediators for interaction with persistent objects of one type. The number of active mediators depends on the current method for mediator management that the XDA uses. A manager extracts *Class ID* and *Marker* from the object reference and uses them for the further dispatching. Here *Class ID* identifies the class of a mediator object. It is used in the second step to find an *extent* corresponding to the mediator. The *Marker* identifies a concrete mediator instance and it is used in the third step to find the right mediator object in the *extent*.

It should be noted that the schema presented in Figure 76 is right only for managers dealing with different types of mediators such as *Object Manager* and *Database Manager*.

Transaction Manager and *Type Manager* are specialised on only one type of mediators and, therefore, using a simplified schema. They skip the second step since the type of mediators is always known.

Performance Estimation

Since the number of available POA's is fixed, performance in the first step can be estimated as $O(4)$.

A relatively small number of classes available in the application server (m) allows several efficient realisations of the second step. In the general case for the indexing here we should use names of classes represented by strings (*Name* in *Class ID*). An average performance in this case can be estimated as $O(c)$ steps if the standard hash table is used. However, if the schema of a database is fixed and will not be changed in runtime, that is usual for many applications, it is possible to avoid the search. Numbering all mediator classes using a fixed in compile time order, we can access them directly by the given index as elements of an array in $O(1)$ time. Saving the index in IPOR (field *Index* in *Class ID*) together with the name of the class in *Class ID*, it is possible to make it persistent and available for request dispatching.

Anyway, since access by the index is very cheap also in combination with the search, it does not drop the total performance significantly. Therefore, in the XDA a combination of both methods was used. First, the manager tries to find an *extent* according to the *Index* and checks the name of the class using the field *Name*. If the name is different, then the manager tries to use the first method and find an *extent* with the help of hash tables by the name.

In the last third step the right mediator object in the *extent* should be found. Similarly, to the Evictor pattern for these purposes the XDA uses hash tables that indexing all active mediators according to their *Markers*. The performance of the search in the hash table depends on the quality of the used hash function and the length of used keys. Let us assume that the standard hash function performs as $O(c_s)$, where c_s is a constant. Since the length of *Object ID* that is used as a key in the standard case is equal to the sum of *Class ID* and *Object Marker* that is used in our approach, the calculation of the standard hash function will take definitely more time. Using smaller keys it is possible to define a more effective hash function that will perform as $O(c_n)$, where c_n is a constant and $c_s > c_n$.

Summarising the estimations for all steps we receive $O(c_s)$ in the worst case, where it is not possible to fix the order of available types, and $O(c_n)$ in the best case. Here we are taking into account only the time that is necessary for the third step of our dispatching approach assuming that the first and the second steps can be performed in $O(1)$ steps.

Since the main idea of the proposed method is to use smaller search keys, the difference in performance depending on the length of used keys was studied in more detail. The following Figure 77 shows performance of hash tables³⁷ on two basic operations that are usually performed during the dispatching of requests. The size of keys is varying from 72 byte (line A) to 2283 byte (line G).

The figures show the difference in performance according to the total number of mediators and their complexity. Here it is easy to see that both presented operations (figures I and II) have almost linear dependency on the size of keys. The difference in performance for the first search operation (figure I) can slow the dispatching of active mediators. However, this drop is not significant comparing it with the degradation of performance for the second operation (figure II), which is performed if a mediator is not active and the eviction should be performed.

³⁷ Here we have used a hash table implemented in ObjectStore's class `os_Dictionary`.

This aspect is also very important for understanding the dispatching of references to complex mediators. Complex mediators are connected with several database objects and their keys are, therefore, correspondingly longer. The exact size of a key depends on the kind of connected database objects and the total number of objects in the database. In this experiment for the construction of complex mediators we have used persistent instances of one class. Different lines in Figure 77 correspond to mediators connected with different numbers of database objects. They start with simple mediators pointing only to one database object (line A) and continue for all following lines with the step in 5 objects.

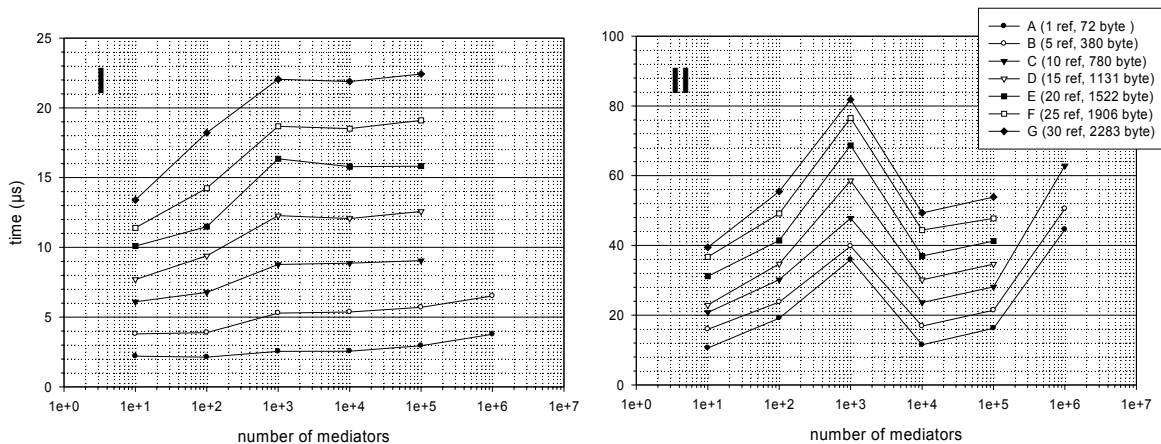


Figure 77. Performance of hash tables on keys of different size: I – search for a key, II – search and replacement of a key (search /remove / insert)

6.3.5 Efficient Eviction Technique based on Reference Substitution

Performance of this search procedure is an important factor but not the one that has a great influence on the overall performance of request dispatching. The other factor is performance of the eviction operation that is performed in the case if the required mediator is not active. It is exactly the place where the standard Evictor pattern significantly loses in performance as it was discussed earlier.

The basic idea of our method is to avoid operations with dynamic memory that are usually performed during activation/deactivation of mediators. Actually, mediators are small objects that have no state except references to the corresponding database objects. Therefore, by grouping active mediators into *extents* that contain only mediators of one type it is possible to replace a pair of operations with dynamic memory (deallocation of the old mediator and allocation of a new one) by one operation for *reference substitution*.

The complete schema for request dispatching in the POA-based XDA using the technique of reference substitution is presented in Figure 78. The separation of all mediators according to their type on several *extents* in the second step of the dispatching allows XDA managers to reuse existing active mediators for serving several requests in the same way as it would be done by a POA with a default servant. By substituting the references, it is possible to use one mediator for serving all requests to persistent objects of the same type.

The procedure can end by three different cases. Initially the manager tries to find an active mediator. Therefore, in the first case the manager can found an active mediator. If the required mediator is not active and cannot be found, then the manager has to decide, if a new mediator object should be created (case II) or an existing mediator will be reused (case III). It is not a simple decision, since the creation of a new mediator object takes more time (`new()` plus reference substitution) than a single operation for reference substitution. Thus, the

creation of a new mediator object can be seen here as an „investment“ in the cache expecting that the object will be requested in the future. However, it is not easy to determine if the mediator is perspective or not in the first access, and which mediator objects are no longer needed. Therefore, an intelligent caching and garbage collection is necessary for the efficient management of mediators. For example, to deactivate mediators which are idle for more than a specified time-out period or when memory runs short.

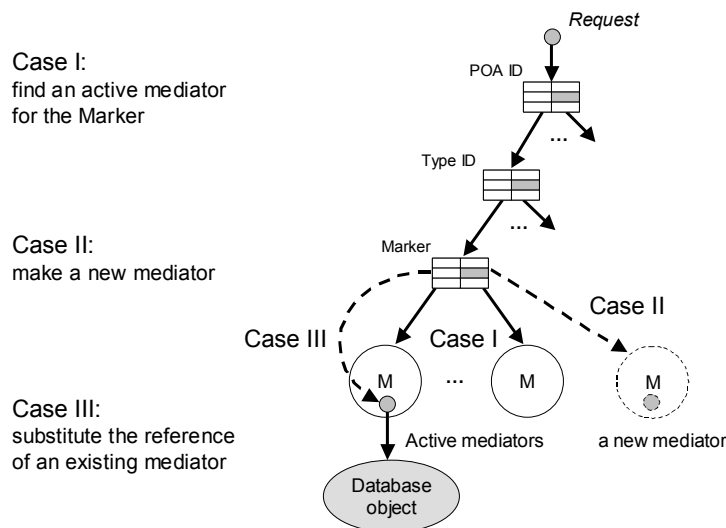


Figure 78. Request dispatching in the POA-based XDA using the reference substitution technique

Performance Estimation

Nevertheless, how much is this difference in performance for different types of mediators? Figure 79 gives the answer to this question comparing the performance of both methods for the eviction of mediator objects. The first (top) line presents performance of the standard approach that is based on the sequence of operations (delete / new / insert reference). The second line presents performance of the new approach performing the eviction by substituting the references. The bottom line is the difference between the both methods.

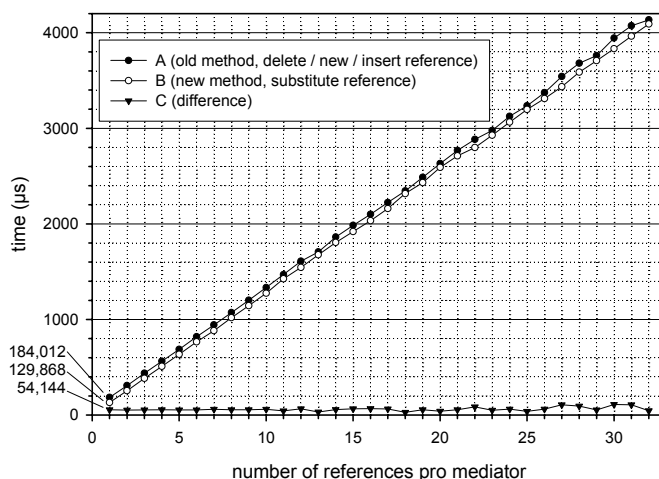


Figure 79. Times for the eviction of mediators depending on the number of associated database objects

From this picture we can see that the difference in 54µs is significant for simple 1:1 mediators comprising about 1/3 of the total dispatching time. For complex mediators that have more

than one reference this difference is not so significant. However, since complex mediators are not so common as simple 1:1 mediators, the method can give a reasonable improvement in the total performance of the dispatching.

6.3.6 Efficient Mediator Replacement Policies

There is a number of effective strategies that can be used for cache management, such as *Least Recently Used* (LRU), *Least Frequently Used* (LFU), evicting the mediator with the highest memory consumption, or using a weighted function that chooses a servant for eviction based on a combination of factors. The goal of this review is to address and discuss the design and implementation aspects specific for caching mediators in the integrated CORBA/ODBMS systems.

6.3.6.1 Formal Definition of mediator's Cache Management

Let us now provide a formal statement for the problem of cache management. Let O be the set of all mediator objects that may be requested in any instance of the time during cache operation. For each object $o \in O$ there a positive size s_o and a positive cost c_o associated with it. The cost c_o is a function F of the following parameters: cost for reference conversion operation r_o , size s_o , time of the last access t_o and frequency of access f_o , that is, $F = F(r_o, s_o, t_o, f_o)$. A sequence of l requests is a function $\Sigma: [1..l] \rightarrow O$ and will be denoted as $\sigma = \sigma_1, \sigma_2, \dots, \sigma_l$. The set of all possible request sequences will be denoted as R . When no cache is used, the cost in servicing sequence σ is $C(\sigma) = \sum_{i=1}^l c_{\sigma_i}$. Let the cache size be X . We will assume that for all objects $s_o \leq X$, that is, no data object is larger than the cache. We define the cache state S_t at time t to be the set of objects contained in the cache at that time.

Definition 1 (The Generalised Caching Problem [Hoss97]).

Let $\sigma = \sigma_1, \sigma_2, \dots, \sigma_l$ be a sequence of l requests resulting in a sequence of cache states S_0, S_1, \dots, S_l such that, for all $S_i, i = 1, 2, \dots, l$,

$$S_i = \begin{cases} (S_{i-1} - E_i) \cup \sigma_i, & \text{if } \sigma_i \notin S_{i-1} \\ S_{i-1}, & \text{if } \sigma_i \in S_{i-1} \end{cases}, \quad (1)$$

where $E_i \subset \Sigma_{i-1}$ denotes the set of items purged out of the cache. Find among all state sequences satisfying Equation (1), a state sequence that minimises the cost

function $F(S_i, \sigma) = \sum_{i=1}^l \delta_i c_{\sigma_i}$ where $\delta_i = \begin{cases} 0, & \text{if } \sigma_i \notin S_{i-1} \\ 1, & \text{if } \sigma_i \in S_{i-1} \end{cases}$ and S_i denotes the state

sequence.

This problem can be viewed both as off-line and on-line depending on how requests are presented to the algorithm. When the request sequence is a priori known then the problem is off-line, otherwise it is on-line. The off-line case of this problem with equals costs and sizes for each object has an optimal solution due to Belady [Bel66] and is the LFD 4 algorithm, which evicts from cache the page whose next reference is furthest in the future. However, non-uniformity in costs and sizes introduces complications and LFD is not optimal any more [Hoss97].

The optimal solution for the generalised caching problem was presented in [Hoss00], which is proven to be in NP. Any solution for the on-line version of the problem must rely on past information in order to make replacement decisions. If some probabilistic information regarding the requests is known, then we can derive some optimal solutions for the on-line problem, as well (see [Hoss97]). But, in practice any solution for the problem (optimal or suboptimal) must be practical, that is, it must perform acceptably well, it must be easy to

implement and should run fast, without using sophisticated data structures and in addition should take into account the peculiarities of the CORBA.

6.3.6.2 Basic Factors

Replacement algorithms deal with the problem of the limited cache space. They try to keep in cache the most „valuable“ data. The „value“ of a datum is usually a function of several parameters, say recency, access frequency, size, retrieval cost, frequency of update etc. The replacement policy makes replacement decisions based on this value. A good replacement strategy should be able to achieve a good balance among all of them and at the same time to „weigh“ differently the most important of them. Depending on these specific factors that we want to consider in the design of a caching policy, (recency, frequency of reference, etc.) we can modify the above definition appropriately and express it as an optimisation problem. For the integrated CORBA/ODBMS environment they are the following.

- time since last access to a mediator
- frequency of access to a mediator
- cost for the loading of a mediator

Since almost all mediator objects are relatively small objects about 100 byte in size, the size factor is not as significant as in common Web applications. The cost for the loading of a mediator is the time necessary for the operation of reference substitution. In the current XDA implementation, this factor is significant only for comparing access times to mediators from different extends. An additional requirement for the policy is the ability to implement it easily without the need of maintaining sophisticated data structures.

6.3.6.3 Performance Measures

The most commonly used performance measures used to characterise the efficiency of a caching algorithm are the *hit ratio*, *byte hit ratio* and *delay savings ratio* [SSV99].

Let O be the set of objects in a cache in a time instance. Let r_o be the total references for object o , cr_o the number of references for object o satisfied by the cache. Let s_o be the size of the object o and g_o be the average delay incurred while obtaining it (reference substitution).

Definition 2 *Hit Ratio* (HR).

The hit ratio of a cache is the percentage of requests satisfied by the cache.

$$HR = \frac{\sum_{o \in O} cr_o}{\sum_{o \in O} r_o}. \quad (2)$$

In essence, improving the hit ratio is equivalent to reducing the average latency seen by a client.

Definition 3 *Byte Hit Ratio* (BHR).

The byte hit ratio (or weighted hit ratio) of a cache is the percentage of bytes satisfied by the cache.

$$BHR = \frac{\sum_{o \in O} s_o * cr_o}{\sum_{o \in O} s_o * r_o}. \quad (3)$$

In essence, improving the byte hit ratio is equivalent to reducing the average traffic of data from the source to the consumer.

Definition 4 *Delay Savings Ratio* (DSR).

The delay savings ratio determines the fraction of communication delays, which is saved by satisfying requests from cache.

$$DSR = \frac{\sum_{o \in O} g_o * cr_o}{\sum_{o \in O} g_o * r_o}. \quad (4)$$

Delay Savings Ratio (DSR) is closely related to *Byte Hit Ratio* (BHR). The latter can be seen as an approximation for the former, where the delay to obtain an object is approximated by its size. Since the size of usual mediator objects in CORBA environment is relatively small, BHR is not applicable for this case.

6.3.6.4 Common Replacement Policies

It is almost impossible to categorise all existing cache replacement algorithms into categories with distinguishable boundaries. Nevertheless, following the proposal of [AWY99], it is possible to categorise the algorithms into three main categories, namely a) traditional policies and direct extensions of them, b) key-based policies and finally c) function-based policies. Here we will discuss only those techniques that are applicable to the integrated CORBA/ODBMS environment.

Traditional Policies and their direct Extensions

The algorithms belonging to the first category comprise direct application of policies proposed in the context of operating systems and databases or modifications of them. The most typical of them are *Least Recently Used* (LRU) and *Least Frequently Used* (LFU). LRU replaces the object, which was least recently accessed. This is the most popular algorithm used today and capitalizes on temporal locality. Its simplicity stems from the fact that in order to make replacement decisions it only needs to maintain a heap with the IDs of the cached objects. Its overhead is $O(n)$ in space (n is the number of cached objects) and $O(1)$ time per access. This is the main reason for its use by many commercial products. An extension to LRU is LRU-K [NNW93], which replaces the document whose k -th reference is furthest in the past. *FIFO* replaces the objects, which entered first in the cache. LFU replaces the object with the least number of references. A variant of the LFU, the LFU-Aging policy [RD90] is much more interesting for CORBA/ODBMS applications since it considers two factors of our interest – the object’s access frequency and its age in the cache (the recency of last access). However, since both LRU and LFU optimise the byte-hit ratio, they were found to be inadequate for CORBA/ODBMS environments.

Key-based Policies

All the above policies suffer from the drawback that they use only a simple characteristic of the cached objects in order to make replacement decisions. For example, LRU uses recency, LFU uses frequency, etc. Alleviating this drawback, key-based policies use a couple or more „keys“ to obtain the objects in sorted order of their „utility values“. The recency of reference, the frequency of reference, etc. can be used as keys by these policies. One of them is selected as primary key, another as secondary key, etc. As replacement victim is selected the object with the least (greatest) value of the primary key. If the first key is broken, then the second and the third keys are used and so on. A representative algorithm of this category is the *Hyper-G* [ASA+96]. It uses the frequency of reference as primary key, the recency of reference as secondary key and the object’s size as tertiary key.³⁸

³⁸ Policies like LRU, LFU, SIZE, can be regarded as function-based policies as well, where the utility function is the inverse of recency, frequency and inverse of size, respectively. They can be regarded as key-based policies as well having only a primary key.

It is evident that key-based policies prefer some factors over others. This may not be always correct. In essence, traditional policies and key-based policies fail to integrate all the relevant factors into a single value that characterizes the utility of keeping an object into the cache. For example, with LRU, the cache can be populated with objects referenced only once purging out documents with higher probability of reference. Consequently, LFU would seem more appropriate, but LFU prevents popular (in the past) „dead“ documents from being evicted from cache and needs an „aging“ mechanism to avoid „cache pollution“ (*LFU-Aging* policy [RD90]). Such a mechanism requires fine-tuning of several parameters and thus it is difficult to implement.

Function-based Policies

Function-based policies assign to every object in the cache a value, „utility value“, which characterizes the benefit of retaining this object in the cache. This „utility value“ is a function of several parameters, such as recency of reference, frequency of reference, size, retrieval cost, etc. Replacement decisions are made using this „utility value“.

It is not possible to separate these policies into disjoint groups, because they incorporate into their „utility value“ different subsets of the set of parameters mentioned above. Nevertheless, we choose to categorise them into three groups, based on whether they capture temporal locality (recency of reference), popularity (frequency of reference) or both. Thus in the first category we have the function-based policies extending LRU, in the second category the function-based policies extending LFU and in the third, the function-based policies integrating LRU-LFU.

Function-based policies extending LRU. The common characteristic of this family of algorithms is that they extend the traditional LRU policy with size and retrieval cost considerations. Their target is to enhance the popular LRU algorithm with factors that account for the special features of the Web. The most popular of them is the *GreedyDual-Size* [CI97], which combines locality, size and latency/cost concerns effectively to achieve the best overall performance. It is a variation of a simple and elegant algorithm called *GreedyDual* [Youn94], which handles uniform-size variable-cost cache replacement.

GreedyDual-Size. The GreedyDual-Size is an elegant algorithm that combines gracefully recency of reference with retrieval cost c_o and size s_o of an object. The „utility value“ associated with an object that enters the cache is the following function.

$$UV_{GD-Size} = \frac{c_o}{s_o}. \quad (5)$$

When replacement is needed, the object with the lowest $UV_{GD-Size}$ is replaced. Upon replacement, the $UV_{GD-Size}$ values of all objects are decreased at an amount equal to the $UV_{GD-Size}$ value of the replacement victim. Upon re-reference of an object o its $UV_{GD-Size}$ value is restored to c_o/s_o . Thus, the $UV_{GD-Size}$ for a cached object grows and reduces dynamically upon re-references of the object and evictions of other objects.

Upon an eviction, the algorithm requires as many subtractions as is the number of objects in the cache. In order to avoid this, we can maintain an „inflation“ value L , which is set to the „utility value“ of the evicted object. Upon re-reference of an object o , instead of restoring its $UV_{GD-Delay}$ to c_o/s_o , we offset this by L . We present the algorithmic form of GreedyDual-Size with the above modification below.

Algorithm 1 (GreedyDual-Size).

Initialise $L \leftarrow 0$

Process each request in turn. The current request is for an object o .

- (1). if o in cache
- (2). $UV(o) \leftarrow L + c_o/s_o$.
- (3). if o not in cache
- (4). while there is not enough cache space for o
- (5). Let $L \leftarrow \min_{q \in \text{cache}} UV(q)$
- (6). Evict q such that $UV(q) = L$.
- (7). Bring o into cache and set $UV(o) = L + c_o/s_o$.

A nice characteristic of this algorithm is its on-line optimality. It has been proved that GreedyDual-Size is k -competitive, where k is the ratio of the cache size to the size of the smallest object. This means that for any sequence of accesses to objects with arbitrary costs and arbitrary sizes, the cost of cache misses under GreedyDual-size is at most k times that under the offline optimal replacement algorithm. This ratio is the lowest achievable by any online replacement algorithm.

The basic disadvantage of this algorithm is that the third available parameter – the frequency of access will be not taken into account.

Function-based policies extending LFU. It is well known [CD73] that when a) the requests are independent and have a fixed probability, and b) the objects have the same size, then the optimal replacement policy is to keep in cache those pages with the highest probability of reference. In other words, the best online algorithm for such applications is the LFU.

Based on this and on the observation that frequency-based policies achieve very high byte hit rates [ASA+96], the function-based policies extending LFU enhance the traditional LFU algorithm with size and/or retrieval cost considerations. The *LFU with Dynamic Aging* (LFU-DA) [DA99] extends the LFU-Aging with cost considerations, whereas the *HYBRID* algorithm [WA97] incorporates size and cost considerations. It is interesting to present how HYBRID computes the „utility value“ of each object in the cache, since it was the first that incorporated the factor of the downloading delay into the replacement decision.

HYBRID. Let an object o located at server s be of size s_o and has been requested f_o times since it entered the cache. Let the bandwidth to server s be b_s and the connection delay to o be c_o . Then, the utility value of each object in the cache is computed as follows.

$$UV_{HYBRID} = \frac{(c_o + \frac{W_b}{b_s}) * f_o^{W_f}}{s_o}, \quad W_f \text{ and } W_b \text{ are tunable constants.} \quad (6)$$

The replacement victim is the object with the lowest UV_{HYBRID} value. Obviously, this algorithm is highly parameterised. New constants W_b and W_f weigh the bandwidth and frequency respectively, whereas c_o and b_s can be computed from the time it took to connect to object o and server s in the recent past.

This family of function-based policies does not include many members, since it does not appear to be a wise choice to ignore the temporal locality in the design of a replacement policy. Therefore, the later efforts concentrated in extending other policies with frequency considerations, for example GreedyDual.

Function-based policies integrating LRU-LFU. This category integrates into the replacement policy both recency and frequency of reference. As expected, these algorithms are more sophisticated than all the previous. The integration of recency and frequency with size and cost considerations results on the one hand in improved performance and on the other hand in having many tunable parameters. As example algorithms of this category we present two

algorithms, namely *Lowest Relative Value* (LRV) [RV00] and *GreedyDual-Delay* – a simple straightforward extension of GreedyDual with frequency parameter.

A replacement algorithm tailored for proxy caches and evaluating statistical information is the *Least Relative Value* (LRV). Statistical analysis of several traces showed that the probability of access for an object increases with the number of previous accesses for it and that the time since the last access is a very important factor and so is the size of an object. All, these parameters have been incorporated into the design of LRV, which strives to estimate the probability of reaccessing an object as a function of the time of its last access, its size and the number of past references for it.

The algorithm evicts the document with the lowest value. The utility value of each cached document that has been accessed i times in the past is computed as follows.

$$UV_{LRV} = \begin{cases} P(1, s_o)(1 - D(t_o)) \frac{c_o}{s_o}, & \text{if } i = 1 \\ P(i)(1 - D(t_o)) \frac{c_o}{s_o}, & \text{otherwise} \end{cases} \quad (7)$$

where $P(i)$ is the probability of re-reference for an object o , given that it has already been referenced i times, evaluated at the time of its last access t_o . The calculation of the $P_i(i, t_o, s_o)$ value is based on extensive empirical analysis of trace data [RV00]. For a given i , let P_i denote the probability that a document is requested $i + 1$ times given that it is requested i times. $P(i)$ is estimated in an online manner by taking the ratio D_{i+1}/D_i where D_i is the total number of documents seen so far which have been requested at least i times in the trace. $P(1, s)$ is the same as $P(i)$ except the value is determined by restricting the count only to pages of size s_o . Furthermore, let $1 - D(t_o)$ be the probability that a page is requested again as a function of the time (in seconds) since its last request t_o ; $D(t_o)$ is estimated as:

$$D(t_o) = c * \log \left(\frac{\beta(1 - e^{-\frac{t_o}{\beta}})}{\alpha} + 1 \right), \quad (8)$$

where parameters c , α , β are computed by the cache on the fly. α and β are in the range of 10-100s and $>5*10^5$ respectively. The schema for adaptive computation of these parameters and $D(t_o)$ is presented in the original publication of this algorithm [RV00].

Essentially, LRV tries to improve hit ratios by taking into account of an object's reference frequency, time of last reference, and size. Apparently, LRV has many parameters that need to be tuned appropriately. This means additional cost and complexity, but LRV can make more clever replacement decisions since it considers more information regarding requests streams. This is a fundamental trade-off: the more information we use, the more efficient our processing is, but this efficiency comes at increased computation cost.

Optimal Cache Replacement Policy for Mediators

Following Figure 80 (a) shows the average hit ratio of the four groups of traces under LRU, Size, LRV, GD-Size(1), and GD-Size(packets). The graphs from left to right show the results for Boston University traces, Virginia Tech traces, DEC-U1 traces and DEC-U2 traces, respectively. Figure 80 (b) is a simplified version of (a) showing only the curves of LRU and GD-Size(1), highlighting the differences between the two.

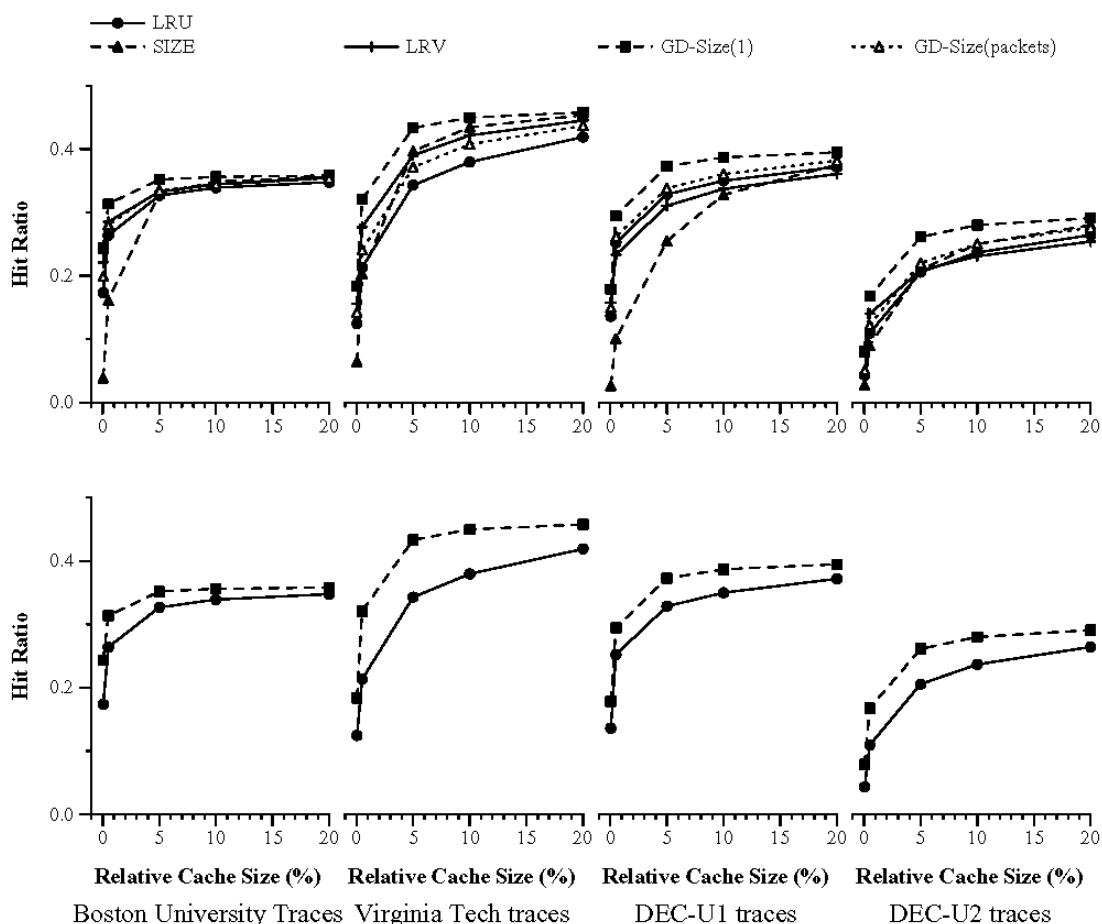


Figure 80. Hit ratio of the four groups of traces under LRU, Size, LRV, GD-Size(1), and GD-Size(packets) [CI97]

The results show that clearly, GD-Size(1) achieves the best hit ratio among all algorithms across traces and cache sizes. It approaches the maximal achievable hit ratio very fast, being able to achieve over 95% of the maximal hit ratio when the cache size is only 5% of the total data set size. It performs particularly well for small caches, suggesting that it would be a good replacement algorithm for main memory caching of web pages.

Least Relative Value (LRV) [RV00] is another replacement algorithm for proxy cache. For the Virginia Tech traces, LRV outperforms GD-Size(packets) in terms of hit ratio and byte hit ratio. This is due to the fact that those traces have significant skews in the probability of references to different sized files, and LRV knows the distribution before-hand and includes it in the calculation. However, for all other traces where the skew is less significant, LRV performs worse than GD-Size(packets) in terms of both hit ratio and byte hit ratio, despite of its heavy parameterisation and foreknowledge.

The LRV algorithm uses a utility function that is based on statistical parameters collected by the server. By separating the cached objects into different queues according to the number of times they are accessed, or their relative size, and by taking into account within a queue only time locality, the algorithm maintains the monotonicity property of LRU within a queue. LRV evicts the best among the objects residing at the head of these queues. Thus, the scheme can be implemented with a constant number of linked lists and find an eviction candidate in constant time. However, its performance is inferior to algorithms like GreedyDual [CI97]. In addition, the cost for maintaining all these linked lists is still high and it has been criticised as heavy parameterisation and high overhead in implementation.

LRU performs better than SIZE in terms of hit ratio when the cache size is small (less or equal than 5% of the total data set size), but performs slightly worse when the cache size is large. The relative comparison of LRU and Size differs from the results in [CI97], but agrees with those in [RV00]. Comparing LRU with GD-Size(1) [Youn94] it is obvious that, for proxy designers that seek to maximise hit ratio, GD-Size(1) is the appropriate algorithm.

However, a big advantage of LRU algorithm is that it can be implemented with a linked list that maintains the order in which the cached objects were accessed so far. This is due to the „monotonicity“ property of its utility function. Whenever an object is accessed, it is the most recently used. Thus, it should be inserted at the tail of the list and the least recently used object always resides at the head of the list. However, most algorithms, including those that have the best hit ratio like GD-Size(1), lack the monotonicity property and they require searching all objects to find which to evict. To reduce computation overhead, they must use a priority queue to drop the search cost to $O(\log k)$, where k is the number of documents in the cache. In particular, Hybrid and GD-Size must use a priority queue. Other problem with GD-Size is that the difference of popularity has not been used.

In this section, we have presented a categorisation of existing cache replacement algorithms applicable for the CORBA/ODBMS environment. Summarising, we can note that the best cache replacement algorithm is in essence the one with the best utility function. We have also presented the most important algorithms belonging to each category. Some of them are relatively simple, whereas some others are more sophisticated. There is no clear „champion“ algorithm, which performs best in all cases. As we will see in section, which discusses replacement issues for some major commercial products, the tendency is to make replacement decisions considering only expiration times and recency of reference. This is because these factors are easy to handle and do not impose high load on the system in order to make replacement decisions. Moreover, they do not require complicated data structures for the maintenance of the metadata associated with cached objects.

6.3.6.5 Implementing the LRU Replacement Algorithm

Searching the best cache replacement strategy for the CORBA/ODBMS environment is a solid research topic on its own that will be saved here. Instead, in this work, implementing a standard LRU algorithm we want to highlight and discuss the aspects specific for CORBA/ODBMS systems. According to our review, the simple LRU algorithm should be effective for the CORBA environment, since it incurs the lowest runtime overhead comparing with other algorithms. Realising this algorithm in the XDA we have investigated its disadvantages and tried to improve its quality by adding additional factors.

To support our approach for reference substitution we have realised the LRU algorithm with the help of two distinct data structures. The first data structure is a hash table that maps *Markers* to C++ mediator pointers and acts as our own active object map. The second data structure we need to implement LRU eviction is a simple queue. Each item on the queue represents a mediator in memory. If the number of active mediators k is small enough ($k < 10$), then it is possible to replace the hash table with a simple linear search in the queue. The main point here is that we must uniquely identify each instantiated mediator with the information in each queued item.

Initially, when the application server starts up, the evictor queue is empty. Whenever a client request arrives, the servant locator's `preinvoke()` operation on the corresponding manager is called, and it first looks in the hash table for the required mediator. If the mediator is already in memory, `preinvoke()` returns a pointer to the mediator. If the mediator is not in memory, the manager should decide either a new mediator object should be created or an existing mediator object should be used (cases II and III presented earlier in Figure 78).

If the manager decides to enlarge the queue, it instantiates a new mediator and adds an entry for the mediator to the hash table, and adds a new item for the mediator to the tail of the queue. Figure 81 shows the evictor queue after `preinvoke()` has been called for the first five objects used by clients after the server start-up. The order of items in the queue indicates the order of instantiation. The item corresponding to the mediator that was instantiated first appears at the top in the queue – that is, as the oldest item.

Let us assume that our queue is limited to holding only five items and that the client sends a request for an object (H), which is not yet in memory. Again, when a new request arrives, the POA calls the `preinvoke()` operation on the servant locator. However, the implementation of `preinvoke()` now realizes that the queue is full. As a result, `preinvoke()` applies reference substitution to the oldest mediator’s item in the top of the queue (1) and moves this item to the tail of the queue. It then replaces an entry in the hash table for the new marker *H*. Now mediator 1 can be used for interaction with database object *H*. The entire process is illustrated in Figure 81.

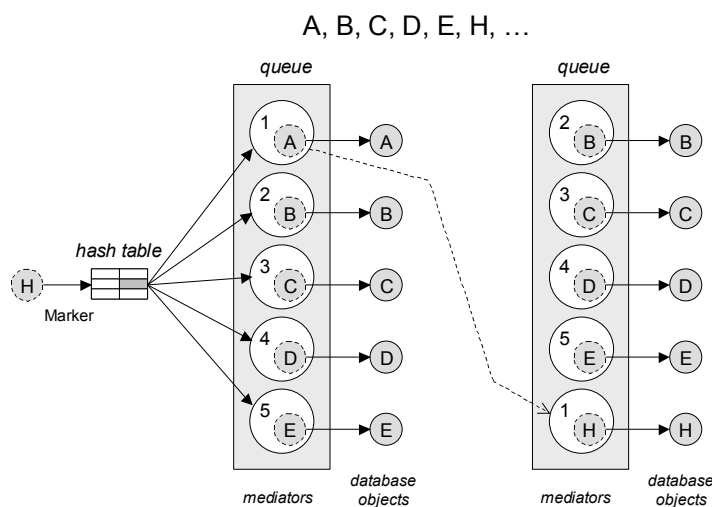


Figure 81. An evictor queue after instantiating five mediators and after eviction of the mediator „1“

The remaining question is how to maintain the queue in LRU order. Conceptually, we want to ensure that every operation that is dispatched to a mediator causes that mediator to be pushed out from its current queue position and to be moved to the tail of the queue. Achieving this goal is simple in our implementation, because the `preinvoke()` operation is called on every request whether or not the mediator is in memory.

- If `preinvoke()` finds a mediator in memory, it moves the mediator’s item to the tail of the queue.
- If `preinvoke()` does not find a mediator in memory, then
 - If the queue can be extended, then it instantiates a new mediator and adds it to the tail of the queue.
 - If the queue cannot be extended, then it applies the reference substitution to the oldest mediator’s item in the top of the queue and moves this item to the tail of the queue.

Either way, each request that is dispatched causes its mediator to move to the top of the queue. With this strategy, after we have located the correct mediator for a request, we must be able to efficiently remove the mediator from the current queue position and insert it at the

top. By storing the queue position of each mediator in hash table, we can locate the mediator on the queue as a constant-time operation. If the number of active mediators k is small enough, then it is possible to drop the hash table and use a simple linear search in the queue. Actually, if we will start the search from the top of the queue, for repeating requests this simple method will give much better performance than hash tables.

If the required mediator is not active and cannot be found, then the manager has to make a decision – if the queue can be extended by a new mediator object or an existing mediator will be reused instead. Analysing reference popularity information for all extents, the XDA manager approximates the size of the extent. Since *Delay Savings Ratio* (DSR) is the most important performance metric for our case, let us try to maximise its value. The DRS is defined as the fraction of communication delays, which is saved by satisfying requests from cache.

$$DSR = \frac{\sum_{o \in O} g_o * cr_o}{\sum_{o \in O} g_o * r_o}. \quad (9)$$

Taking into account that for every object in an extent O_i the cost for the operation of reference substitution g is a constant, there is m extents and the queue of i -th extent O_i has N_i objects, we receive the following equation.

$$\sum_{o \in O} g_o * cr_o = \sum_{i=1}^m g_i * cr_i, \quad (10)$$

The task for maximisation of DSR function can be transformed to the minimisation of the back function. Assuming that the number of successful requests cr_i for a particular extent O_i is proportional to the length of its queue N_i we can maximise DSR extending the queue proportional to the number of misses. Here the specific aspects of our cache are taken into account, since the objects of different extents (classes) have different distributions of access probability. Let us see the following example of a client program.

```
// C++
W_Part_var pVar;
XDA_Database_var dbVar = ...;
XDA_Collection_var collVar = ...;

for (int i=0; i<1000; i++) {
    pVar = W_Part::_narrow(dbVar->create_object("W_Part"));
    pVar->id(i);
    collVar->insert(pVar);
    ...
}
```

Here a database object `dbVar` plays the role of the factory that is invoked a number of times to create a set of other `W_Part` objects. The number of referenced `W_Part` objects is equal to the length of the circle and can be very large comparing to the number of objects representing collections `XDA_Collection`. Extending the cache proportional to the number of misses, we receive the following equation.

$$\begin{cases} \Delta N_i \approx g_i \frac{r_i}{cr_i} \\ \Delta N \approx \sum_{i=1}^m g_i \frac{r_i}{cr_i} \end{cases} \Rightarrow \Delta N_i = \frac{g_i \frac{r_i}{cr_i}}{\sum_{i=1}^m g_i \frac{r_i}{cr_i}} \Delta N. \quad (11)$$

Collecting the popularity information for all extends the XDA manager enlarges the size of a particular extent ΔN_i proportionally to all extents ΔN . While the distribution of the number of objects with at least i accesses for one particular extent is unique, summarising the lengths of all extents we can expect the same distribution as it was experimentally estimated for web traces as presented in Figure 82.

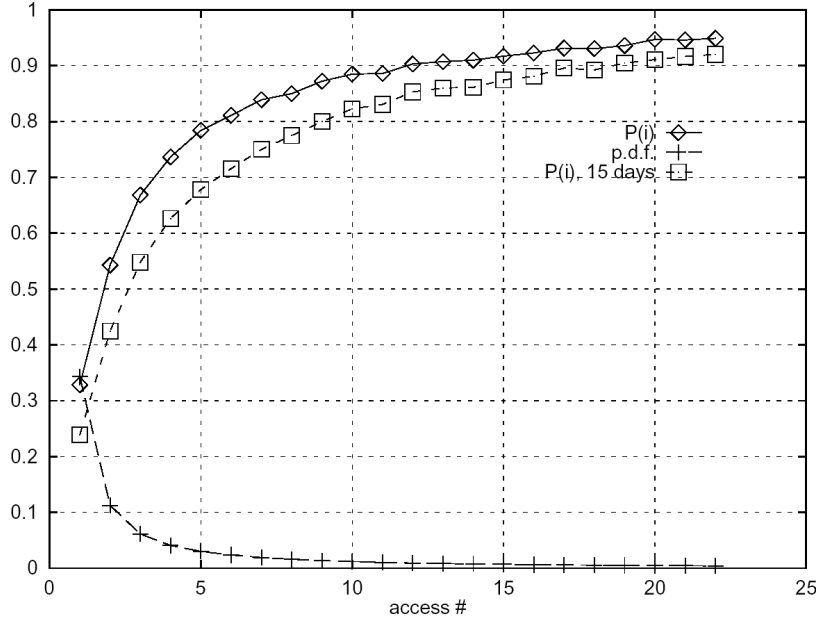


Figure 82. Top curve: the probability of more accesses after i previous ones ($P(i)$). Middle curve: $P(i)$ computed without considering new accesses issued after 15 days. Bottom curve: percentage of documents with at least i accesses [RV00]

In particular, assuming that the number of objects references at least i times is roughly proportional to $1/i$. Let $K(f)$ is a function giving the number of objects from the number of accesses. Gathering statistic data for two points $K(1)=k_1$, where k_1 is the number of object with one access and $K(f_{max})=k_{max}$, where k_{max} is the number of objects with at least f_{max} accesses, we can approximate the function as follows.

$$K(f) = \frac{a}{1+bf} = \frac{(1-f_{max})k_{max}k_1}{k_1 - k_{max}f_{max} + (k_{max} - k_1)x} \quad (12)$$

The total number of objects in the cache can be estimated as an integral of the approximation function $K(f)$ from 1 to f_{max} .

$$K(k_1, k_{max}, f_{max}) = \int_1^{f_{max}} K(f)df = \frac{k_{max}k_1(f_{max} - 1)}{k_{max} - k_1} (\ln(k_{max}(1 - f_{max})) - \ln(k_1(1 - f_{max}))) \quad (13)$$

Calculating $K(k_1, k_{max}, f_{max})$ and comparing its value with the total number of objects in the cache, it is possible to make a decision when the size of the cache should be extended. If the value of $K(k_1, k_{max}, f_{max})$ is larger then the total number of objects N , then a new mediator object can be created. In the other case, an existing mediator should be reused.

Here function $K(k_1, k_{max}, f_{max})$ according to the collected reference popularity information for cached objects describes the „quality“ of the cache helping to avoid the pollution phenomenon, to which we alluded earlier. The following Table 16 presents values of $K(k_1, k_{max}, f_{max})$ for some points. We can see that if the number of mediators with only one

access k_i grows not proportionally to the number of objects with more accesses, then the values of K are going down and, thus, limiting the total number of objects in the cache.

N	k_i	k_{max}	f_{max}	$K(k_i, k_{max}, f_{max})$
11	10	1	10	23.02
101	100	1	100	460.52
1001	1000	1	1000	6907.75
11	10	2	5	16.09
101	100	2	50	391.20
1001	1000	2	500	6214.60
21	20	1	10	28.3
31	30	1	10	31.66
41	40	1	10	34.05
21	20	2	5	20.5
31	30	2	5	23.2
41	40	2	5	25.2

Table 16. Values of $K(k_i, k_{max}, f_{max})$ for some points

6.3.6.6 The Problems of LRU Replacement Policy

The *Least Recently Used* (LRU) replacement policy is widely used due to its simplicity. However, the observations reflect LRU's inability to cope with access patterns with weak locality such as collection scanning, regular accesses over more objects than the cache size, and accesses on objects with distinct frequencies [JZ02]. Here are some representative examples reported in the research literature, to illustrate how LRU poorly behaves.

1. Under the LRU policy, a burst of references to infrequently used objects, such as „sequential scans“ through a large collection, may cause the replacement of commonly referenced objects in the cache. This is a common complaint in many commercial systems: sequential scans can cause interactive response time to deteriorate noticeably [NNW93]. A wise replacement policy should prevent „hot“ objects from being evicted by „cold“ objects.
2. For a cyclic (loop-like) pattern of accesses to a collection that is only slightly larger than the cache size, LRU always mistakenly evicts the objects that will be accessed soonest, because these objects have not been accessed for the longest time [SKW99]. A wise replacement policy should maintain a miss rate close to the cache space shortage ratio. Practically these problems can be illustrated on the following example of a client program, which performs a simple circle.

```
// C++
XDA_Part_var pVar = ...;
XDA_Database_var dbVar = ...;
XDA_Collection_var collVar = ...;

for (int i=0; i<1000; i++) {
    pVar = W_Part::_narrow(dbVar->create_object("W_Part"));
    pVar->id(i);
    collVar->insert(pVar);
    ...
}
```

It differs from the previous example that new objects are created with the help of an object of the same class. Since the number of new objects can be large, they are able to „pollute“ the cache using a simple LRU algorithm, probably popping the object

partVar out. This phenomenon is analogous to thrashing whereby a popular, newly cached object is evicted before building up enough „inertia“ (in terms of access frequency) to resist eviction due to a burst of references to an object that is popular only over a short time scale [GB97]. The situation is exacerbated further as cached objects age – the longer the request stream, the larger the „inertia“ needed to resist eviction. Since the access frequency of a new popular object is always computed from scratch, it has no fair chance to stay in the cache unless its reference popularity information is maintained even when the object itself is temporarily evicted.

3. In an example of multi-user database application [NNW93], each record is associated with a B-tree index. There are 20,000 records. The index entries can be packed into 100 blocks, and 10,000 blocks are needed to hold records. We use $R(i)$ to represent an access to Record i , and $I(i)$ to Index i . The access pattern of the database application alternates references to random index blocks and record blocks by $I(1)$, $R(1)$, $I(2)$, $R(2)$, $I(3)$, $R(3)$, Thus, index blocks will be referenced with a probability of 0.005, and data blocks are with a probability of 0.00005. However, LRU will keep an equal number of index and record blocks in the cache, and perhaps even more record blocks than index blocks. A wise replacement should select the resident blocks according to the reference probabilities of the blocks. Only those blocks with relatively high probabilities deserve to stay in the cache for a long period.

The reason for LRU to behave poorly in these situations is that LRU makes a bold assumption – an object that has not been accessed the longest would wait for relatively longest time to be accessed again. This assumption cannot capture the access patterns exhibited in these workloads with weak locality. Generally speaking, there is less locality than that in CPU caches or virtual memory systems [RD90].

However, LRU has its distinctive merits: simplicity and adaptability. It only samples and makes use of very limited information – recency. However, while addressing the weakness of LRU, existing policies either consider more history information, such as *Least Frequency Least* (LFU)-like ones in the cost of simplicity and adaptability, or switch temporarily from LRU to other policies whenever regularities are detected. In the switch-based approach, these policies actually act as supplements of LRU in a case-by-case fashion. To make a prediction, these policies assume the existence of relationship between the future accesses to an object with the behaviours of those objects in its temporal or spatial locality scope, while LRU only associates the future behaviour of a block with its own history references.

Summarising, the plain LRU algorithm has a number of problems.

- How to capture the temporal locality exhibited in object access streams, although it is weak?
- How to avoid cache pollution – the tendency of previously popular objects to linger in the cache?
- How to maintain efficiently the popularity profile of a large working set of objects?
- How to use such a profile to estimate the long-term access frequency of individual objects more accurately?

6.3.6.7 Improving the LRU Replacement Algorithm

For our implementation of the reference dispatch in the XDA, the size of objects and the cost for the operation of reference substitution for all objects in one extent is constant. In fact, here we have two varying parameters: recency and frequency of access. Therefore, taking into account the frequency of requests it is possible to improve the standard LRU algorithm. Let us name it as *LRU+*.

Using statistic data for web traces from Figure 82 let us design a utility function UV estimating the probability that the object can be accessed next time. Calculating this function for objects referencing more than one time, it is possible to prevent their eviction from the cache. Thus, if the probability that the object will be accessed next time is more than $\frac{1}{2}$, then it is probably more wise to keep this object and give it a second chance.

Let an object o been requested f_o times since it entered the cache and the time from its last access t_o . Then, the utility value of each object in the cache is computed as follows.

$$UV_{GD-FT} = P_r(f_o, t_o). \quad (14)$$

$P_r(f_o, t_o)$ is the probability that the object can be accessed next time. Approximating this function in the similar way as in [RV00], it can be calculated as a product $P_r(f_o, t_o) = P(f_o) * P(t_o)$. If we look at $P(f_o)$ at Figure 82, we see that its value grows significantly with f_o , up to 0.9 and more for $f_o > 12$. Hence, the number of previous accesses appears to be a good indicator of the probability of a new access. $P(f_o)$ function can be approximated with the following parameterised function.

$$P(f_o) = 1 - \frac{1}{1 + \alpha * f_o}, \quad (15)$$

where the parameter α here determines the periodicity of frequent references to popular objects. For the presented distribution, it is about 0.4. However, using this parameter alone to determine $P(f_o)$ is a bad idea, because this does not permit the aging of documents, something that is absolutely necessary in order to avoid pollution of the cache.

To account for this, a mechanism must be adopted to give preference to more recent references in prediction. In our implementation, we use a decay function to de-emphasise the significance of past accesses according to the interaccess time. Denning and Schwartz [DS72] established the fundamental properties that characterise the phenomenon of temporal locality. Such properties are the catalyst for well-established practices in the design of caching systems in hierarchical memory structures [Smit82]. In order to apply these practices to the design of caching systems for CORBA/ODBMS environments, it is important to characterise the degree of locality present in typical request streams. Since there is no information about CORBA-based streams available, we can use estimations made for Web access streams.

Using an independent reference model [CD73] in [BCF+99] it was showed that the Zipf-like popularity distribution of objects in Web request streams could asymptotically explain other properties (namely, cache efficiency and temporal locality). In particular, they showed that the probability of referencing an object in t_o times after it has been last accessed is roughly proportional to $1/t_o$ (Figure 82). Thus, the probability distribution of reference interarrival times can be used to model temporal locality. It turns out that the probability density function of interaccess times can be reasonably approximated with k/t_o . In particular, on the t_o -th access to an object o , the probability distribution $P(t_o)$ can be approximated as:

$$P(t_o) = \frac{1}{1 + \beta * t_o}, \quad (16)$$

where β is a parameter accounts for the long term decay in the range (0, 1). Estimating the probability that the object will be reaccessed in 10 sec as $\frac{1}{2}$ we receive $\beta = 0.1$. The following Figure 83 illustrates both approximation functions.

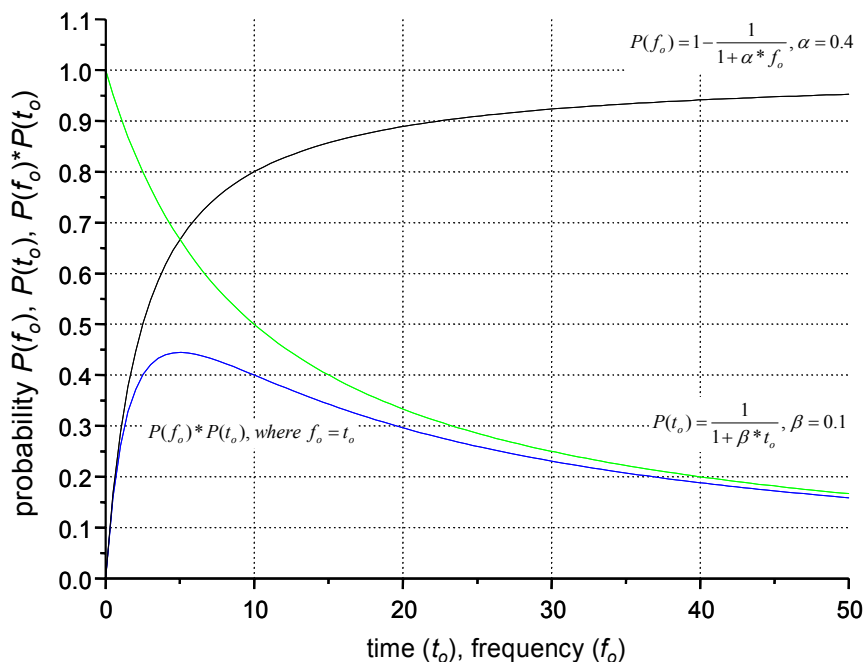


Figure 83. The curves represent probabilities that the object will be reaccessed after f_o previous ones – $P(f_o)$ and after time t_o – $P(t_o)$. The bottom curve is the product of the both functions P for $t_o=f_o$ ³⁹

As a result, the number of previous accesses and interaccess times may be used for calculating the utility function.

$$UV_{GD-Delay} = \begin{cases} \left(1 - \frac{1}{1 + \alpha * f_o}\right) \left(\frac{1}{1 + \beta * t_o}\right) \\ 1/2, \text{ for new objects} \end{cases} \tag{17}$$

In some cases, an object with many past accesses can have a higher probability of being reaccessed than a much more recent object with just one or two accesses. The use of $P(f_o)$ does not change the LRU ordering within classes of objects with the same number of accesses. As the largest set of objects those with one access belongs to the same class, we would like to make further distinction among them basing on some other parameter. Objects with only a single access are inserted in the middle of the queue that is equal to $1/2$ probability of the following reaccessing.

6.3.6.8 Garbage Collection of persistent CORBA Objects in general

In the most general sense, garbage collection is the automatic removal of resources that are no longer in use by a program, without explicit action by the application code. In CORBA, garbage collection refers to reclamation of resources that are used by objects that are no longer of interest to clients. So far, we have considered garbage collection only of transient mediator objects. This arrangement simplifies the problem considerably because, in this case, the mediator and the CORBA object are usually the same thing; they are both created at the same time and destroyed at the same time using servant managers or somewhat similar. When we consider persistent CORBA objects, however, it becomes difficult even to decide what garbage collection should, mean, let alone how to implement it.

³⁹ Origin[®] mathematical software was used for approximation of statistic results.

Consider a persistent CORBA object representing a person. The object could keep a record of personal details in a database. Clearly, person objects have long life cycles, typically measured in decades. If we want garbage collection for CORBA objects, we must decide what garbage collection actually means. In particular, if we have a persistent object representing a person and decide to garbage-collect the object, the question arises whether to destroy only the corresponding mediator object or to destroy the persistent database record of the person also. For persons, the answer is most likely „no“. After all, the fact that we have reclaimed a mediator to free some memory does not mean that the corresponding person has died.

Looking at objects with a shorter life span, it becomes more difficult to make a clear-cut decision about whether to destroy only the mediator or both, the mediator and the database object. For example, we may have document objects that represent documents in an archive. Typically, when clients „lose interest“ in a document, we want to reclaim the mediator for a document object but retain the persistent state of the document. On the other hand, sooner or later, the statute of limitations expires and we want to garbage-collect not only the mediator for the document but also its persistent state. To make matters worse, when the time has come to destroy the document, there may be no mediator in existence to remind us that it is time to destroy it.

The second important point here is the problem of *referential integrity* and dangling references. Because CORBA permits persistent references to propagate by uncontrollable means, there is no way of knowing whether an IPOR is still of interest to some client or not. In the preceding scenario, it is very difficult to recognise which persons are still interested in a particular document. The life time of IPORs is not limited and, for example, there after many years can be still some persons interested in the document that have received it's IPOR from other persons. An equivalent action is to write a stringified reference into a file, to send a stringified reference in an e-mail message, or to bind a reference in the Naming Service. The semantics of persistent references make it impossible to safely garbage-collect an object. We could decide to destroy an object at any time without warning, but that might leave a client with a dangling reference. The next time that client used the reference, it would get an `OBJECT_NOT_EXIST` exception.

The preceding examples show that the meaning of garbage collection is highly dependent on each application's requirements. In some cases, to garbage-collect an object means to reap its mediator. In other cases, both mediator and persistent state must be destroyed, and in yet still others, the circumstances change over time.

6.3.7 Evaluation of the new Dispatching Technique

The POA-based XDA provides a common framework for implementing different deactivation (eviction) strategies. Since the proposed method for mediator management combines the best sides of two wide used methods – the Evictor pattern and default servants, it is reasonable to expect corresponding improvement in performance of request dispatching. Using this technique, we are able to serve several servants simultaneously as it does evictor pattern and at the same time to keep their number relatively small as it is with default servants.

6.3.7.1 Advantages and Trade-Offs of Reference Substitution

Similarly, to the standard Evictor pattern, maintaining a structure of type-oriented pools of active mediators the proposed method allows simultaneous processing of several requests and in the same time having a good performance on repeating requests. The other strong side of this method is scalability. The problems with dynamic memory that are typical to the

Evictor pattern are solved using idea of default servants as a general eviction policy. Using this technique it is possible to reduce the number of active mediators to assist scalability of the application server on large databases.

The following Table 17 summarises the costs for the standard Evictor pattern and the proposed method for reference substitution. In the following discussion we will refer to them as the first and the second methods.

	Evictor pattern	Reference substitution
Storage	one mediator per database object	one mediator per class of database objects
Search	$O(c_s)$, where c_s is a constant	$O(c_n)$, where c_n is a constant and $c_n < c_s$.
Replacement	three operations – delete(), new(), set_reference()	only one operation for reference substitution – set_reference()

Table 17. Comparison of the standard Evictor pattern with the proposed method for reference substitution

Table 17 shows that the first method saves the server's memory when the server must deal with many classes of mediator objects. It uses one large pool of n elements for active mediator objects of all types. In the contrast, the second method uses a tree-like structure from several pools. If it is m classes and k is an average number of active mediators per class, then the second method requires memory only for $m*k$ active mediators. Minimally using only one active mediator per class, it is possible to go out with m active mediators totally. If the number of classes m is quite large, then using the first method it is possible to limit the total number of active mediators by an n , which can be much more smaller then m . However, taking into account that the number of classes is usually smaller then the number of active mediator objects, it is usually possible to find space for m active mediators at least.

Comparing both approaches, we can see that the first method saves the server's memory when the server must manage with many types of mediators. On the other hand, it loses in performance in general case, because activation of a new mediator object always assumes implicit deactivation of another object. In the case that the evictor query is full, most operation invocations from clients cause substitution of the reference for an existing mediator, and that is much more cheaper than corresponding three operations if one mediator instance should be evicted and another mediator to be instantiated. When a client has two objects of the same type and by turn accesses them, almost every request causes an invocation of the loader. It can be very inefficient, for example, in the case when a client navigates through a large collection of objects.

Using a technique similar to default servants, the second approach has better scalability then the first, because a single mediator object can be used to represent an unlimited number of CORBA objects simultaneously. As a result, the mediator itself is stateless, which makes this approach ideal if a CORBA server is used as a front end to a database that may be updated independently by many clients.

6.3.7.2 Experimental Results

Unfortunately, this theoretical comparison is very inaccurate and the only real way to find out what an adapter is really capable of is to create a benchmark. The cost of call dispatch varies considerably among environments and depends on a large number of variables, such as the underlying network technology, the CPU speed, the operating system, the efficiency of the TCP/IP implementation, the compiler, and the efficiency of the ORB runtime itself.

Therefore, to give a rough idea of the XDA's dispatching efficiency we have made some benchmarks that show memory consumption and the performance of dispatching for BOA- and POA-based versions of the XDA. For both versions we implemented a test application, which is described in Section 7.3.1.

Orbix 2.3c [ITorbix97] and *ORBacus 4.0.2*⁴⁰ [ITorbacus01] from IONA Tech. were used as the BOA- and POA-capable ORB implementations, respectively. All programs were coded in C++ and compiled with Sun's C++ compiler version 5.2 without either debugging information or optimisation.

The results were obtained by running a client and a server on the same Sun SPARC Ultra 2 workstation (xenon). It had 640 megabytes of memory and was configured with two SPARC processors running at 400 MHz. *ObjectStore* version 6.0 SP8 from ODI Inc. [ODIug01] was used as an ODMG-conformant DBMS. The size of the ObjectStore's client cache manager was set to 16777216 bytes.

Dispatcher benchmarks for BOA- and POA-based XDAs with the unlimited cache are presented in Figure 84, I and Figure 84, II, respectively. The test consists of a sequence of the following operations.

1. `create_children` – creates a test set of `XDA_Part` objects.
2. `retrieve` – retrieves the references to the created objects. This is different for BOA- and POA-based versions, since in the latter version only an object reference is generated. The BOA-based version creates a real mediator object.
3. `ping` – a simple operation that does nothing. This operation was called twice to see how much time the creation of a new mediator would take. In the POA-based version it takes about 2 msec (Figure 84, II). In the BOA-based version, the mediators are already created and activated with the previous operation `create()`. Therefore, both operations `ping()` take the same time.
4. `delete` – deletes all created objects and deactivates all corresponding mediators.

This sequence was performed on variable numbers of persistent objects. All operations were made within separate manual transactions. Simple operations iterating through all sets of objects such as `ping` were grouped together and performed within one common transaction.

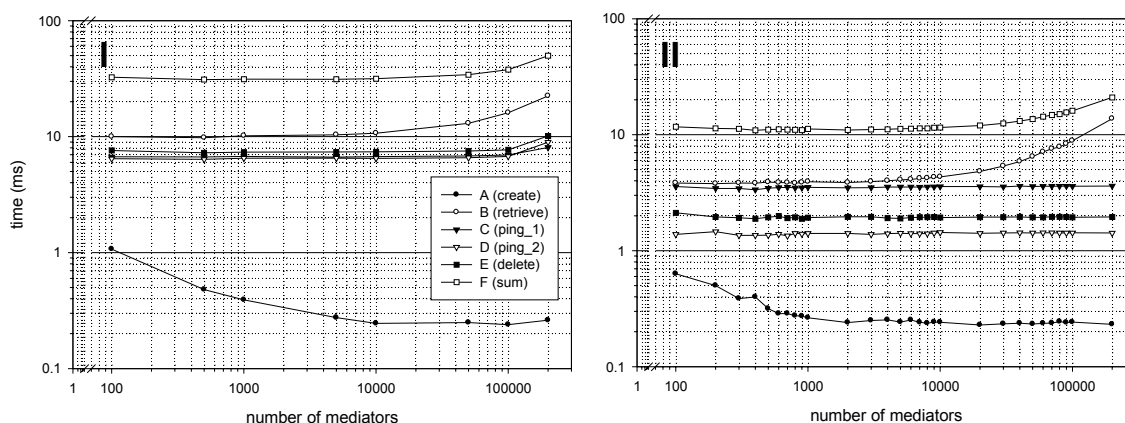


Figure 84. Performance of particular operations for BOA- (I) and POA-based XDAs (II) with unlimited cache

⁴⁰ Initially developed by OOC Inc., now ORBacus is distributed by IONA Tech.

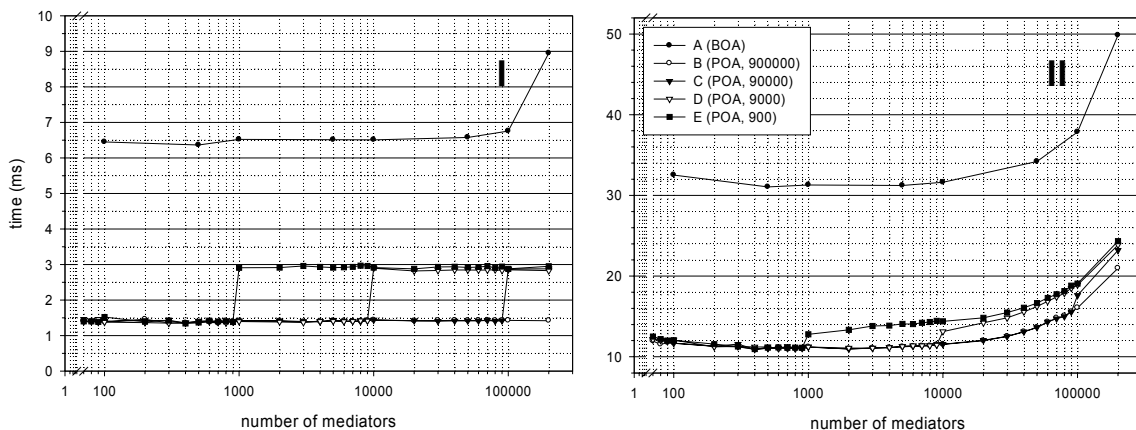


Figure 85. Performance of request dispatching in BOA- and POA-based XDAs for different cache sizes:

- I – average call latency for operation ping (second),
- II – average time for a total sequence of operations (create, retrieve, ping, ping, delete)

Figure 85, I presents the performance of request dispatching operations in the BOA- and POA-based versions. For the latter version, the test was repeated with different cache sizes: 900000, 90000, 9000 and 900 active mediators (lines B-E). Here we can see that the new approach realised in the POA-based version gives average call times for ping operations of between 1.4 msec for active mediators, and 2.9 msec otherwise. In other words, on this hardware and using this ORB implementation, it is reasonable to expect a maximum call rate of 714 to 345 remote operation invocations per second.

The „jumps“ at 1000, 10000 and 100000 mediators marked for the POA-based version (lines B-E) can be explained by the caching effect, which happens when the total number of active mediators exceeds the size of the cache used by the object manager. The difference in times of 1.5 msec is the time that is necessary for the object replacement operation. Compared to the total dispatching time that is between 1.4 and 2.9 msec, we see that this is a relatively expensive operation.

However, the request dispatching times for the POA-based version are relatively small when compared with the result for the BOA-based version (line A), in which the dispatching operation takes about 6.5 msec (only 154 calls per sec). The difference between the times required for the dispatching operations of the BOA and POA-based implementations is 5.1 msec (a factor of 4.6x) in a case of active mediators, and 3.6 msec (a factor of 2.2x) otherwise.

Figure 85, II presents the time for the complete sequence of operations (create_children, retrieve, ping, ping, delete) showing the impact of the evaluated call dispatch operations on the total time. As pointed out earlier, to determine the call dispatch cost for a particular environment, appropriate benchmarks needs to be performed.

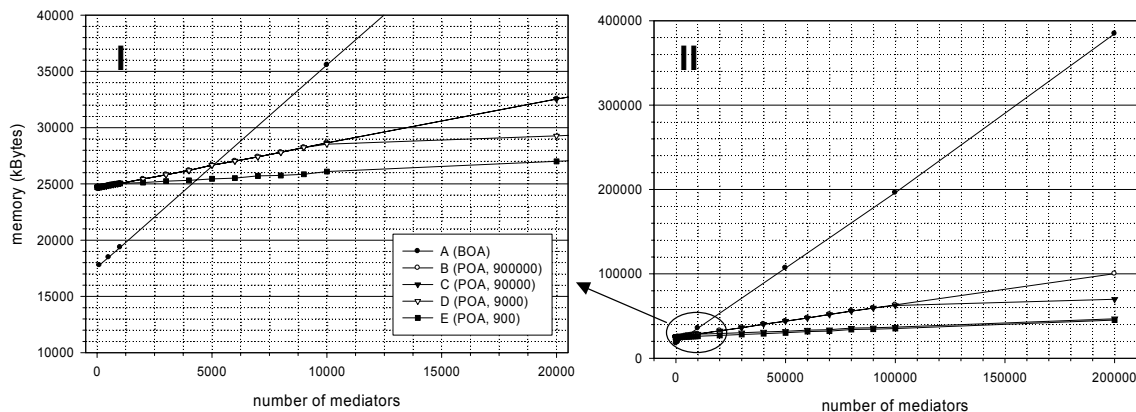


Figure 86. Memory consumption in BOA- and POA-based XDAs for different cache sizes. Figure I represents a zoomed part of figure II in the region 0..10000

The last Figure 86 shows the memory consumption for both XDA versions. Figure 86, I represents a zoomed part of Figure 86, II in the region 0..10000. Here we see that using default servants, the scalability of a CORBA server is no longer limited by its memory consumption and depends only on the bandwidth to the database and the number of parallel invocations that can be supported. The unlimited scalability of the adapter comes at a price though: each access to the database using reference substitution technique takes longer than using multiple active mediators. However, if the accesses have good locality of reference and the database has effective caching, the performance penalty of the reference substitution can be surprisingly small, so the technique is well worth exploring. It has been shown that intelligent caching can provide precise control over the memory consumption versus performance trade-off for the request dispatching.

The results show that the proposed technique for request dispatching provides a reasonable compromise between the common techniques of servant managers and of default servants by compensating one with the other. On the one hand, by preventing frequent operations for memory allocation/de-allocation, application of this technique significantly reduces the fragmentation of dynamic memory and, therefore, improves the stability of the integrated CORBA/ODBMS system. On the other hand, by managing several active mediators per class, the method does not require the serialisability of the requests that is typical for default servants allowing simultaneous processing of several requests. The reference substitution technique can help servers to achieve high performance without consuming massive amounts of memory and retains strong stability.

6.4 New Techniques for Transaction Management

Every act of reading or writing to the database must be within a transaction. An XDA transaction represents a unit of work that is handled by the XDA’s transaction manager and determined by the user or by the adapter. The model assumes two basic interaction modes for controlling transactions: *automatic* and *manual*.

The evaluation of the transaction management implemented in the BOA-based version of the XDA on geoscientific applications shows some weak points of both modes. To eliminate these defects of transaction management, new approaches for transaction processing were developed. They extend transactional models implemented in the BOA-based version by

proposing new transaction's types and employing new algorithms for maximising their safety and performance.

6.4.1 Problems with Automatic and Manual Transactions

1. Performance problems on applications using large numbers of small operations.

Appears using *automatic per-operation* transactions.

Using automatic *per-operation* transactions on applications making many remote calls follows to the corresponding number of necessary transactions. In this case, operations for starting and finishing transactions can take more time than an operation that was called.

2. Performance problems on databases that are simultaneously used by multiple clients.

Appears using *automatic delayed* and *manual* transactions.

Manual and automatic *delayed* transactions can lock the database for very long time following to bad concurrency in the multi-client environments.

Persistent objects that are accessed by a client within a transaction are locked for all other clients. They can access these objects only upon the running transaction commits. In the case of one client, this feature gives a definitive advantage, reducing the overhead for frequent starting and committing transactions. However, in the case of multiple clients these savings are often outweighed by the overhead from waits for locks to be released.

3. Problems with rollbacks and deadlocks.

Appears using *automatic delayed* transactions.

Automatic delayed transactions are introduced to solve the overhead introduced by using large numbers of many small transactions. However, this optimisation breaks traditional ACID principles and, therefore, has serious problems with rollbacks and deadlocks.

On rare occasions, the ObjectStore system may abort a transaction either because of a network failure or because it has determined that the transaction is involved in a deadlock. The CORBA client that uses automatic *delayed* transactions does not know when the current transaction is started and, therefore, cannot repeat actions, which are rolled back.

The same problem appears in the case of deadlock. A simple deadlock occurs when one transaction holds a lock on a page that another transaction is waiting to access, while at the same time this other transaction holds a lock on a page that the first transaction is waiting to access. Neither process can proceed until the other does. By default, the transaction is restarted at the first statement following the start of the transaction, but if it was delayed, it will be not possible. See simple deadlock scenario in Section 6.4.2.1. Other, more complicated forms of deadlock are analogous.

4. Exhausting of persistent address space

Appears using *automatic delayed* and *manual* transactions.

Transfers of large amounts of data in the database using only one transaction can cause segmentation faults.

The granularity of server transfers can be changed from a page to a segment, with the latter resulting in en masse transfer of a complete segment. For each transaction, only those parts of the database(s) that are accessed by the transaction are mapped into the address space of the client. This strategy introduces a restriction on the total amount of data that can be referred to by any single transaction. Large operations need to be broken down into a series of smaller transactions.

6.4.2 Multiclient Concurrency Control

Concurrent access of a database by multiple clients, which is enabled by CORBA's client/server architecture, creates the potential for actions of one client to interfere with another. One application can reduce the waiting overhead for other concurrent applications by avoiding locking data unnecessarily and by avoiding locking data for unnecessarily long periods. However, what can we do to reduce the overhead of waiting for database locks?

This section describes several techniques used in the XDA for minimising the wait time in the case of concurrent access, for example.

1. The XDA allows to perform read-only transactions through nonblocking *Multi-Version Concurrency Control* (MVCC) reads of the database. This will allow another ObjectStore application to update the database concurrently, with no waiting by either the reader or the writer.
2. To avoid locking data for unnecessarily long periods using delayed transactions, the XDA realises a simple communication protocol for exchanging information about blocked objects between application servers.
3. Using custom object clustering, CORBA clients can reduce wait time for locks. By default, each page is locked as it is faulted on; the default locking granularity is page level. CORBA clients or XDA managers can override this default specifying cluster-, segment-, or database-level granularity.

6.4.2.1 Using Multi-Version Concurrency Control (MVCC)

ObjectStore offers a concurrency control option called *Multi-Version Concurrency Control* (MVCC). This will enable applications to open a database in a read-only mode that allows MVCC transactions to access the database without ever waiting for a lock or being the victim of a deadlock.

The POA-based version of the XDA implements MVCC operations on the IDL interface of the adapter. When a CORBA client uses MVCC, it can perform non-blocking reads of a database, allowing another client or ObjectStore application to update the same database concurrently, with no waiting by either the reader or the writer. Therefore, if a CORBA client does both of the following, it should open the database for MVCC.

- Only performs read access on a database.
- Does not require a view of the database that is completely up to date, but can instead rely on a snapshot of the data.

In each transaction in which an application accesses a database opened for MVCC, the application sees what it would see if viewing a snapshot of the database taken sometime during the transaction. If an application accesses databases that have been opened for multidatabase MVCC it will see snapshots of those databases taken at the same moment in time. The snapshot has the following characteristics.

- It is internally consistent.
- It might not contain changes committed during the transaction by other processes.
- It does contain all changes committed before the transaction started.

A good way to update an application's snapshot is performing a checkpoint.

The basic advantage of MVCC is that an application using a database opened for MVCC has never to wait for locks to be released in order to read the database. Reading a database opened for MVCC also never causes other applications having to wait for the update of the

database; see the example MVCC and the Simple Waiting Scenario. In addition, an application never causes a deadlock by accessing a database it has opened for MVCC. See the example MVCC and the Simple Deadlock Scenario.

Transaction Locking Examples

The following examples illustrate some of the locking situations described in this chapter.

1. Simple Waiting Scenario.

Usually if one transaction (Transaction 1) reads a page *p* and then another transaction (Transaction 2) reads the same page, the second transaction is not blocked. However, if the second transaction tries to write to the page, it must wait until the first transaction commits. If one transaction reads a page of a database it has opened for MVCC and then another transaction attempts to update the same page, the second transaction is not blocked.

Usual		MVCC	
Transaction 1	Transaction 2	Transaction 1	Transaction 2
read <i>p</i>		read <i>p</i>	
	read <i>p</i>		read <i>p</i>
	write <i>p</i> : (blocked)		write <i>p</i> : (succeeds)
Commit		commit	
	write <i>p</i> : (succeeds)		

Table 18. Schedules for usual and MVCC transactions in the simple waiting scenario

2. Simple Deadlock Scenario.

In the first schedule below, Transaction 2 attempts to write *p1*, but cannot proceed until Transaction 1 completes and releases its read lock on *p1*. However, Transaction 1 cannot proceed until Transaction 2 completes and releases its lock on *p2*. Because neither transaction can proceed until the other does, the result is a classic deadlock scenario. ObjectStore chooses Transaction 1 as victim and aborts it, whereupon Transaction 2 can proceed.

In the second schedule below Transaction 1 uses a database opened in the MVCC mode. Transaction 2 writes *p1* without waiting; it can proceed before Transaction 1 completes and releases its read lock on *p1*, because Transaction 1 has the database containing the page opened for MVCC. Similarly, Transaction 1 can proceed before Transaction 2 completes and releases its lock on *p2*. Using MVCC it is possible to avoid the deadlock resulted in the first schedule.

Usual		MVCC	
Transaction 1	Transaction 2	Transaction 1	Transaction 2
read <i>p1</i>		read <i>p1</i>	
	read <i>p1</i>		read <i>p1</i>
	read <i>p2</i>		read <i>p2</i>
	write <i>p2</i>		write <i>p2</i>
	write <i>p1</i> : (blocked)		write <i>p1</i> : (succeeds)
read <i>p2</i> : (blocked) – deadlock		read <i>p1</i> : (succeeds)	
abort			
	write <i>p1</i> : (succeeds)		

Table 19. Schedules for usual and MVCC transactions in the simple deadlock scenario

Performance Notes

Although MVCC can eliminate locking delays that can occur in regular, non-MVCC transactions, they can still cause an application to wait for the resolution of a callback. In most cases, the resulting delay is only a matter of milliseconds. Such a delay can occur under the following conditions.

- An MVCC client reads a page for which an update client has write ownership, or an update client acquires a write lock on a page for which an MVCC client has read ownership.
- A snapshot has not already been taken.
- The other client does not have a lock on the page that the server knows about.

In this situation, a callback occurs to check whether there is a conflict that requires a snapshot to be taken. There is no conflict if the page is not locked and thus it can be called back. Delays related to callbacks can occur in either single-database or multidatabase MVCC.

In addition, when a multidatabase MVCC snapshot is being taken concurrently with the commit of an update transaction, and the intersection of the snapshot's data with the update transaction's written data spans more than one server, a delay may occur while the servers determine whether the update is or is not visible in the snapshot. In most cases, this delay is unnoticeable, but in the worst case it may take a few seconds. Of course, if all the multidatabase MVCC databases opened by a given client are hosted by one server, a delay can never arise.

This delay can slow either the application server performing MVCC or the update application server and is not under XDA control. It depends on the load of the server. Network communication delays that are usual in the case of traditional ObjectStore application are usually eliminated, since the XDA application server, which at the same time is an ObjectStore's client, is located in the same machine as ObjectStore's database server.

6.4.2.2 Using Notifications

One way to avoid locking data for unnecessarily long periods is to make transactions as short as possible while still guaranteeing that persistent data is in a consistent state between transactions. Applying this technique to „delayed“ transactions, the XDA transaction manager finishes a long-lived transaction, if there are some other clients waiting for a lock on an object that was associated with the transaction. The ObjectStore notification service is used in the XDA to notify other application servers that some locked objects are now required. This feature allows the XDA to manage the delayed transactions in the way to minimise the waiting time if a conflict occurs.

Typically, a notification event corresponds to the modification of an object in a database, but applications can assign their own meanings to events. In our case, it is an event signalling that a required object is locked. The notification is generated by an application server in the case if an implementation of an IDL operation accesses a persistent object, which is locked by the other process. The lock probe is made by the XDA's object manager resolving the reference to the object (`operation os_ref()`) first time.

A notification, however, is always associated with a single database location or with a range of database locations. To separate these service notifications from custom database notifications creation of a new XDA-specific location is required. Therefore, all new databases created with the XDA have an additional service root named as `XDA_Notification_Root`. If an XDA application server opens a database, it automatically subscribes to notifications occurred at this location. Subscriptions are stored in the ObjectStore server as long as the

corresponding database is opened by the client. The notification broadcasts to all subscribers that an event (for example, blocking) has occurred at a location (for example, XDA_Notification_Root) in one of the opened databases. Since every XDA application server is subscribed to the location XDA_Notification_Root in all opened databases, it receives only notifications from the database that it opened.

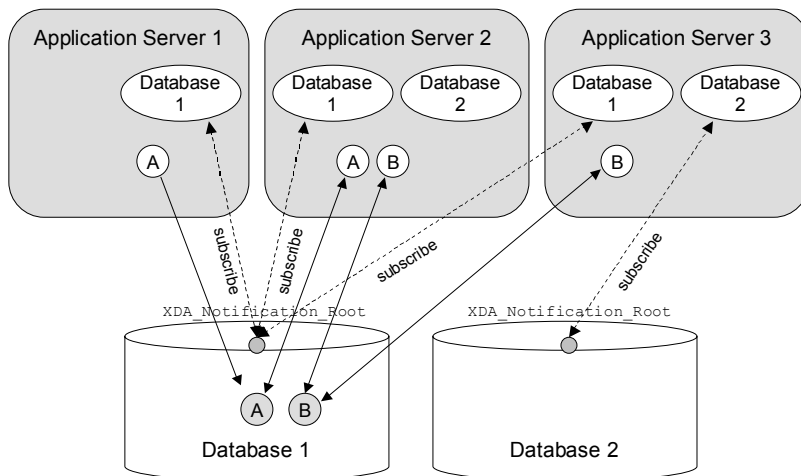


Figure 87. Using notifications for controlling delayed transactions

When a notification is posted, a message is sent to the ObjectStore server. The server then matches the notification with subscriptions and queues messages to be sent to the receiving processes. If there are no subscribers, the notification is discarded. Notifications proceed asynchronously to the cache manager of the receiving application servers.

Using the notification service, three XDA application servers send and receive events about locks to be released as soon as possible. The delayed transaction will be committed as short as possible while still guaranteeing that persistent data is in a consistent state between transactions. An example of using notifications for controlling delayed transactions is presented in following figures. Figure 87 shows the architecture of the example and Figure 88 presents a sequence diagram.

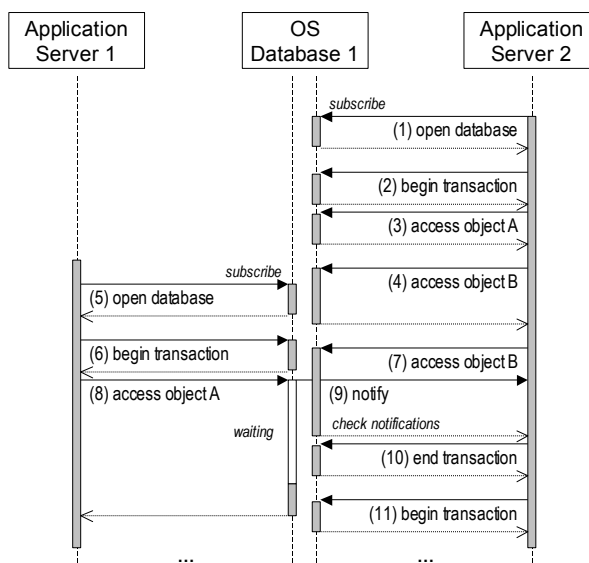


Figure 88. Sequence diagram for an example of using notifications presented in Figure 87

Let us study a case where three application servers (1, 2, 3) work with two databases (1, 2). All application servers subscribe to the notifications for the location `XDA_Notification_Root` in all opened databases. The application server 2 has started a delayed transaction and has locked an object *A*. The application server 1 also wants to access the object *A* and tries to get a lock, but recognises that somebody has already locked the object. In this case, the application server 1 sends a notification and waits. Finishing the current IDL operation on an object *B*, the application server 2 checks for notifications from other servers available and commits the transaction. The lock on object *A* is released and the application server 1 can now access the object.

The proposed solution for controlling length of delayed transactions has not only advantages, but also some drawbacks. Main drawbacks of this schema are the following.

- Releasing gathered locks is coupled with the end of IDL operations. Since now the XDA uses a passive single-session processing model, there is no possibility to finish an active transaction, if a corresponding CORBA client is not active for a moment.
- The model is inaccurate. All application servers that have opened a given database will receive the notification and will finish their transactions as soon as possible. Returning to the previous example (Figure 87), we see the case where the third application server that does not use object *A* should also finish its current transaction.
- The usage of the ObjectStore notification service for communication between application servers brings an additional overhead. All notifications and subscriptions on a database go to the ObjectStore server. The server routes notifications to interested clients, and the cache manager queues the notifications for all its clients. Notifications are stored and forwarded in the server, in the cache manager, and sometimes even in the receiving application. Therefore, delivery of notifications might not be particularly fast. Performance varies according to system load and the amount of notification processing. For example, delivery could range from milliseconds to several seconds. As a rule, the throughput is not high and can be not more than approximately ten notifications per second.

6.4.2.3 Using Clustering for Reducing wait Time for Locks

One way to help avoid locking data unnecessarily involves the use of clustering. Suppose that during a given transaction, an application requires object *A* but not object *B* (Figure 87). If the two objects are clustered onto the same page, they are both locked, preventing other processes from accessing both objects until the end of the transaction. In contrast, by clustering object *B* in a different object cluster or segment from object *A*, a client can guarantee that the objects will be on different pages. Therefore, if the default page-level locking granularity is used, the objects are not locked together.

Using custom object clustering, CORBA clients can also reduce wait time for locks. By default, each page is locked as it is faulted on, since the default locking granularity is page level. CORBA clients or XDA managers can override this default with the `set_lock_option()` member of `XDA_DbCluster`, `XDA_DbSegment`, `XDA_Database`, or `XDA_Adapter`, specifying cluster-, segment-, or database-level granularity.

Overriding the default avoids the overhead of a separate page fault for each page locked and can reduce conflicts between concurrent application servers. However, bad clustering can also lock data unnecessarily and increase wait time.

6.4.3 Managing available Resources

To manage the memory that an ObjectStore application uses, it is necessary to understand the principles of how the memory is organized. An ObjectStore application uses three kinds of memory.

- Physical memory is the real memory (RAM) available on the machine. All applications running on the machine share the real memory.
- Virtual memory contains the application's data. Each application has its own virtual memory.
- Persistent memory is that in which application's persistent data is stored. Each application has its own persistent memory.

ObjectStore transfers an application's data to virtual memory. Persistent memory is a subset of virtual memory. It is just a place where the ObjectStore client manipulates persistent data accesses within a transaction.

Address space is a range of virtual memory addresses that an application can use. For most 32-bit computers, the address space is slightly less than 2^{32} bytes. Each client process has its own address space. The address space is distinct from the virtual memory.

- Address space includes all addresses that could be assigned. It does not matter whether the address is in use. It typically is larger than virtual memory.
- Virtual memory includes only addresses that currently contain application data, either in physical memory or in backing store on disk.

Since an ObjectStore application can have two kinds of data – *persistent data* that the application accesses in an ObjectStore transaction, and *transient data* that are static or allocated in the stack or heap. Client's address space has accordingly two kinds of addresses.

- *Persistent address space.* This is a range of addresses that ObjectStore uses to store persistent data only.
- *Transient address space.* This includes all addresses in the address space except those designated for persistent data.

Both arts of application's data are stored in virtual memory. It does not matter whether the data has a persistent address or a transient address. When a page of data needs to be brought into physical memory, the operating system often needs to make room by removing another page. When the operating system removes a page from physical memory, it places the page on disk in the appropriate backing store. This means that at any particular time data can reside in one of the following.

- Physical memory
- Backing store
 - The client cache, which is the backing store for persistent data
 - Swap space, which is the backing store for transient data.

If the page contains persistent data, the operating system pages it to the client cache. Each client has its own cache. If the page contains transient data, the operating system pages it to swap space. Swap space is a file or disk partition that is shared by all applications running on the host.

The following Figure 89 shows address space. The persistent storage region is near the middle of the address space. The stack allocates transient data starting at one end of the range

of addresses. The heap allocates transient data starting at the other end of the range of addresses. The stack and the heap can allocate space until they reach the persistent storage region.

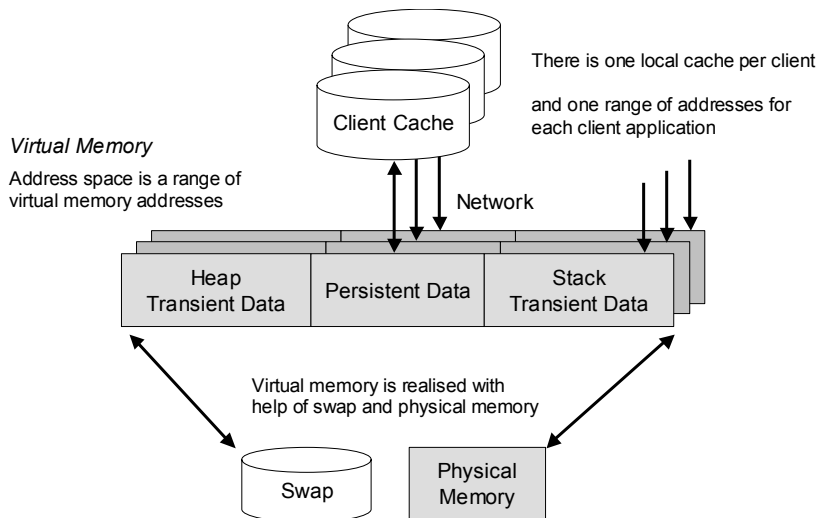


Figure 89. Organisation of memory in ObjectStore applications

6.4.3.1 How do Transactions use Address Space?

Address space is always reserved in 64 KB units. When address space is reserved for an object, ObjectStore actually reserves addresses for the entire 64 KB-aligned portion of memory containing the object. This document refers to a 64 KB-aligned portion of memory for which address space is reserved as a *reservation chunk* or, simply, *chunk*.

As a transaction proceeds, the amount of address space reserved typically increases. As more pages are mapped, more chunks typically are referred to by mapped pages. As more chunks are referred to by mapped pages, more address space is reserved. By default, ObjectStore releases (that is, unreserves) all reserved address space at the end of each transaction. Therefore, address space limits the amount of data a program can touch within a single top-level transaction. Address space must be reserved to relocate data correctly into virtual memory. Space is reserved as each page is used in the transaction, and the amount of space is the minimum required to relocate the page correctly into virtual memory.

Under some circumstances, on 32-bit platforms, address space reservation can consume all address space in the course of a single transaction. This might happen (on 32-bit platforms), if the pages used by a transaction refer to chunks whose total size approaches the total size of the application’s persistent storage region (or, for applications that use the sessions facility, the total size of the current session address space partition). When all address space is reserved, ObjectStore signals `err_address_space_full`.

Another unpleasant situation can have place with swap space, since virtual memory is used for both, for transient and persistent data. If swap space is needed but is not available, the operating system does one or both of the following.

- Prohibits other processes from starting.
- Aborts the process that required additional swap space.

Often a process is aborted when it tries to allocate more memory than it is available. If an ObjectStore allocation of persistent memory fails in this way, an exception is signalled (`err_insufficient_virtual_memory`). When the whole address space is reserved, why does ObjectStore not automatically release some, just as it automatically evicts least recently

used pages when the client cache is full? The answer concerns pointer validity. When pages are released from address space, all encached pointers to those pages become invalid, because the virtual memory locations designated by the pointers can now be reserved for different data. Any automatic eviction scheme makes it difficult or impossible for the program to determine the pointers that have been rendered invalid.

6.4.3.2 Controlling Address Space Reservation

Address resources are large and it is unusual to exhaust them. However, the lack of a particular resource might affect an application. Therefore, to prevent unexpected exhausting of memory resources, the XDA takes control over them.

For applications that want to care about the amount of address space ObjectStore gives two basic possibilities to do that.

- Keep transactions short.
- Release address space explicitly in mid-transaction, before it is all reserved or handle `err_address_space_full`.

A simple solution to the problem of address space exhausting could be to handling the exception `err_address_space_full`. Upon the exception an application can release address space explicitly. It can be realised, for example, using markers or the operation `objectstore::release_persistent_addresses()`.

This method is effective, but it is applicable only locally in mediators for particular functions. In the general case the XDA cannot determine the pointers that have been rendered invalid and we receive here the same problem with pointer validity as it was mentioned earlier. With explicit address-space release, the mediator, in effect, can specify the pointers to be released. So there is no question about the pointers that have been rendered invalid and the program can take care not using the pointers again. If the program subsequently needs pointers to the same objects, it knows to re-retrieve the pointers from the database.

Example of explicit Address Space Control in Mediators

Suppose that a function must process a large vector of pointers and that the pointers point to objects scattered among many different reservation chunks. XDA's transaction manager manages transactions automatically and the function is processed in a delayed transaction. Further, suppose that after an element of the vector has been processed, it is not used again in the transaction.

To avoid exhausting address space, the mediator can release address space at various points during the traversal of the vector. In the worst case, each vector element points to a different reservation chunk. In this case, the mediator should release address space after each `RES_LIMIT` element has been processed. `RES_LIMIT` is the size of the persistent storage region divided by 64 kb (minus the number of chunks occupied by addresses reserved before the marker and minus a few chunks for ObjectStore internal data).

If there are approximately 100 referents of the vector elements in each reservation chunk, the application can process 100 times as many elements before releasing address space. Here is an example of releasing address space during a vector traversal.

```

XDA_BEGIN_TXN (tx1, XDA_Transaction::update)

    Foo *p = ...; // needed across release() calls
    os_address_space_marker the_marker;
    Foo *vec1[LARGE_NUMBER] = ...;

    for (int i = 0; i < vec_size; i++) {
        if ( (i-1) % RES_LIMIT == 0 )
            the_marker.release();
        process (vec1[i], p);
    }

XDA_END_TXN (tx1)

```

Automatic Controlling the Size of the Address Space

Since the XDA has no control over the execution process in midtransaction, it uses the other method to prevent exhausting resources by controlling the sizes of reserved and available address spaces. By default, the whole reserved address space is released automatically at the end of each transaction. Therefore, controlling the length of delayed transactions by breaking the long transactions into multiple shorter transactions, each of which consumes less address space, XDA manages the consume of address space in general.

If XDA's transaction manager thinks that a delayed transaction is too long and the next IDL operation might exhaust the address space, it finishes the current transaction. This decision will be made in the point where the transaction can be finished and is based on three factors: the time the transaction runs, the number of performed IDL operations and the amount of unallocated available address space. To determine if there is an unreserved address space portion of a given size available, the XDA uses an `ObjectStore` operation `objectstore::is_unassigned_contiguous_address_space()`. However, the potential problem here is that the XDA cannot automatically determine the size of necessary address space. The amount of memory a particular IDL operation needs depends entirely on the application. Here the mediator's developers, which understand the application's data structures and data access patterns, should cooperate with XDA's transaction manager setting the right size of unreserved address space.

6.4.4 Chained Transactions

Automatic transactions are controlled by the adapter transparently to the client's activity. Each style has its strong as well as weak points. The *per-operation* style can incur considerable overhead when many methods of relatively low cost are invoked. The *delayed* style removes this overhead violating traditional ACID principles and only guaranteeing that access to persistent data will take place and it will be done always within a correct transaction.

On the other hand, when the number of clients is high, long-lived transactions negatively affect the scalability of the overall system. The *per-operation* style does not have this problem, but is too costly. Hence, both solutions are not acceptable for real applications and new techniques to transaction management are necessary. For example, it would be beneficial to develop a technique that somehow combines the advantages of both styles: the safety of *per-operation* style with performance of the *delayed* style.

By performing a checkpoint operation on a transaction, we can commit modified data without incurring some of the usual overhead of ending a transaction and starting a new one. This principle is followed to the development of a new style: *chained transactions*.

6.4.4.1 Using Checkpoints to improve the Safety of delayed Transactions

Checkpoints represent an alternative solution to commit for continuing automatic transactions. By performing a checkpoint operation, the transaction manager can commit modified data without incurring some of the usual overhead for ending and starting a new transaction. A checkpoint operation

- commits the current top-level transaction.
- immediately starts a new transaction.
- immediately acquires read locks on all or most persistent objects that were locked before the checkpoint.
- retains the validity of pointers to persistent memory across the checkpoint boundary. Note that address space is not released across checkpoint boundaries. See Section 6.4.3.2.
- refreshes the application's view of any database opened for MVCC. See Section 6.4.2.1.

Therefore, the basic advantage of checkpoints against commits is that this operation is generally less expensive than acquiring the locks incrementally, as it happens without checkpoint in an explicitly started transaction. Any locks used before the checkpoint will be reacquired as read locks only as long as there are no another clients waiting for a write lock on an object that was associated with the current transaction. If another client is waiting for a write lock on a persistent object that was locked in the current transaction, performing a checkout the application loses that lock. If there were any write locks before the checkpoint, ObjectStore changes them to read locks or gives them to any clients waiting for those write locks. Consequently, the client that made a checkpoint might have to wait for locks or might get a deadlock when it tries to update the database again.

6.4.4.2 Using chained Transactions for long-lived Operations

The interesting question here is whether or not a certain transaction model is adequate for the processing requirements of long-lived computations such as updating one million accounts. Let us return to our example with one million of bank accounts. Now assume again the bulk of the one million updates is decomposed into a sequence of a million chained transactions. Comparing to simple *delayed* transactions, chaining improves the situation considerably, because whenever a failure occurs, it is only the last transaction (i.e., the ongoing IDL operation) that is rolled back. In this case, a CORBA client can resume the normal processing returning to an IDL operation that returned an exception about the rollback. At the same time due to checkpoint operations, *chained* transactions provide better performance than *per-operation* transactions.

Although chained transactions are recoverable, they still do not guarantee atomicity for the overall computation. The reason is simply that when the chain of transactions is being processed and something goes wrong, the responsibility of the transaction system at restart is to make sure that the updates of all complete transactions in the chain are not lost, and that the ongoing transaction that was interrupted by the crash is rolled back.

There is another important difference between a sequence of individual transactions and a huge flat transaction: a flat transaction that updates all account records as one ACID unit of work makes all effects visible at the same point in time, namely at its commit. A chain of transactions releases the updated values gradually; there are as many commit points as the chain has elements. Updating all accounts in one ACID unit ensures that no intermediate update occurs, such as inserting a new account record while the application is still running. If

the computation is broken up into a sequence of separate transactions, this guarantee cannot easily be maintained.

6.4.5 Evaluating the proposed Techniques

To find out how heavy the influence of transaction management is on the performance of the integrated CORBA/ODBMS system and to evaluate the effectiveness of new transaction management mechanisms, a set of measurements has been performed. The presented tests were performed using a test application that also was used for other tests presented in this thesis. The detailed description of the test application can be found in Section 7.3.

6.4.5.1 Test Environment

The results were obtained running a client and a server on the same Linux workstation (thror). It had 1 gigabyte of memory and two Pentium III processors running with 1 GHz. The CORBA's server program was also written in C++ but compiled with the GNU C++ compiler version 2.95.3.

All programs were compiled without including debug information and any optimisations. *ORBacus* 4.0.2⁴¹ [ITorbacus01] from IONA Tech. was used as the ORB implementation. *ObjectStore* version 6.0 SP8 from ODI Inc. [ODIug01] was applied as an ODMG-conformant DBMS. The size of the ObjectStore's client cache manager was set to 16777216 bytes.

6.4.5.2 Experimental Results

Performance of integrated CORBA/ODBMS applications heavily depends on the way the mediators are managed and, especially, the way transaction boundaries are handled. First, we investigated the effects of automatic transaction styles on different kind of remote operations. The presented test consists of a sequence of operations: `create`, `set_id`, `set_id`, `set_data`, `multiply_read_good`, `multiply_write_good`, `delete`, which were performed on variable numbers of persistent objects. Figure 90 shows performance values of particular operations for different transaction styles: I – *manual*, II – *perOp*, III – *delayed* (1trx), IV – *delayed* (1trx, checks). Figure 91 presents the same test results, but in slightly different projection. The last Figure 92 shows times for the complete sequence of operations.

The first trend that we noticed is the overhead of the code running in a transactional context. When the „perOp“ mode is enabled (figure II), the overhead on iterating operations such as `set_id` (lines B and C) is impressively large. It has a factor of 3x, comparing with the case where manual transactions are performed. This result can be explained by the overhead for starting and committing a separate physical transaction for every IDL operation. Figure 91, I presents all these times in a single picture.

The delayed style represents the upper bound on performance benefit that might be achieved (Figure 90, III). In this test all operations are performed inside of one update transaction. Our measurements show significant gains relative to the manual transactions on complex operations, such as `multiply_read_good` and `multiply_write_good`. They are presented in Figure 91, II. The „jumps“ at 4000 mediators happen, because the size of all accessed objects exceeds the size of the cache used by ObjectStore's client manager. In the cases, when all accessed objects are in the cache, a single transaction reduces the overhead for their processing to a factor of 4x.

However, since the CORBA client does not control delayed transactions, it also does not know where the current transaction is started and, therefore, cannot repeat the actions in the

⁴¹ Initially developed by OOC Inc., now ORBacus is distributed by IONA Tech.

case of a rollback. The ability to roll the transaction back to a safe state and to restart a transactional operation is a crucial capability for providing adequate recoverability of data. Addressing this problem, we have designed a mechanism of *chained transactions*, which provides correct transactional rollback for automatic delayed transactions. Each IDL operation within the chained transaction is finished by its own checkpoint, and the standard rollback semantics apply. All changes made by the operation are automatically rolled back to the previous checkpoint (IDL operation) when a particular operation aborts.

Figure 90, IV shows performance of such chained transactions. We see that in general this style of transactions performs like *perOp* (Figure 90, II) and the difference between them is not significant. Only on iterating operations such as operation *set_id* the test shows 2x improvement in the performance (Figure 91).

Looking at times for the complete sequence presented in Figure 92, it is easy to make the final statement that chained transactions are not as effective as manual transactions. However, since the majority of transactional processing with the XDA will be done likely in the automatic mode, on the one hand, this style represents an interesting alternative for the *perOp* style. On the other hand, although chained transactions are safe, their performance is worse than those for simple delayed transactions. It must be because of a large number of checkpoints. For simple database-powered CORBA applications the performance can be more important than the safety. Therefore, for such applications the XDA provides a compromise solution – automatic delayed transactions making fewer checkpoints. The XDA transaction manager decides when such a transaction should be finished and how. For example, if a checkpoint will be enough or the transaction must be committed. But what factors can be used to determine that?

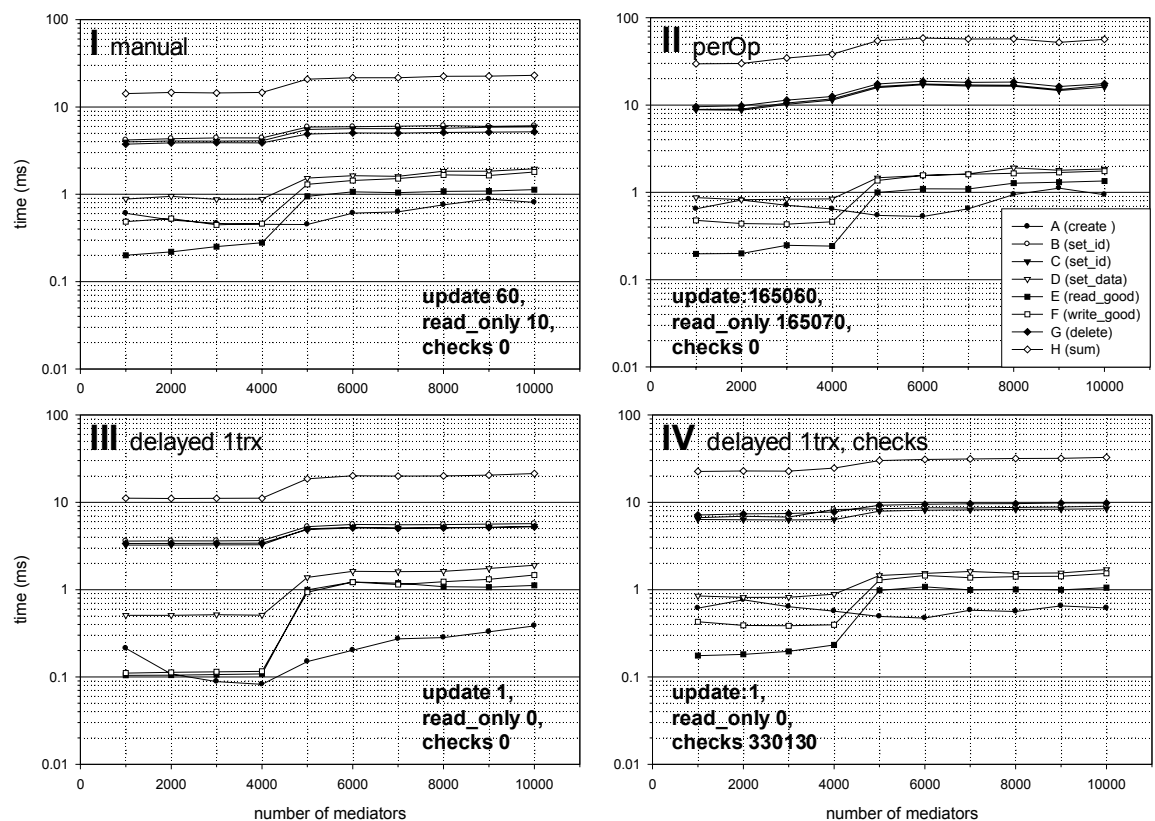


Figure 90. Performance of the test application using different transaction styles: I – *manual*, II – *perOp*, III – *delayed* (1trx), IV – *delayed* (1trx, checks)

The XDA transaction manager uses the following basic factors that are available.

- *Time.* If a transaction is active for a long time ($t > t_{max}$), it should be finished. In this case, a checkpoint will be enough.
- *Number of IDL operations.* Similarly to the time, this factor shows the length of the transaction. If the number IDL operations is large enough ($n > n_{max}$), the transaction should be finished. In this case, a checkpoint will be enough.
- *Available resources.* Long transactions will be terminated by a commit operation, if the size of available persistent address space goes low.
- *Notifications from other clients.* The delayed transaction will be terminated by a commit operation, if there are some other clients available that are trying to access some objects locked by the transaction.

In the current version of the XDA these parameters can be set by the client. Figure 91 and Figure 92 present performance of such a transaction that makes checkpoints in every 60 sec (line E). It has totally only 17 checkpoints and the performance approaching those for the simple delayed transactions (line C). Thus, the benefits of this style can be seen only when multiple CORBA server processes concur for access to persistent objects stored in database.

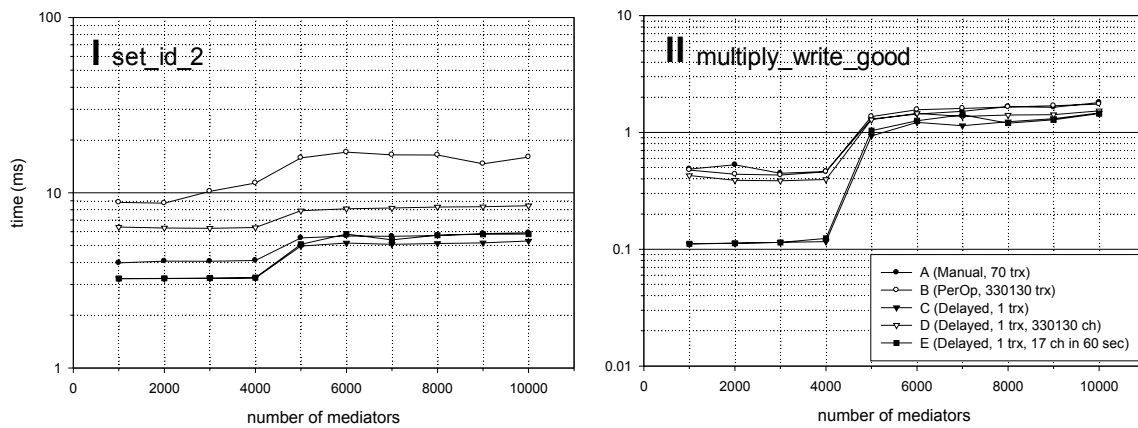


Figure 91. Performance of particular operations using different transaction styles: I – set_id (second), II – multiply_write_good

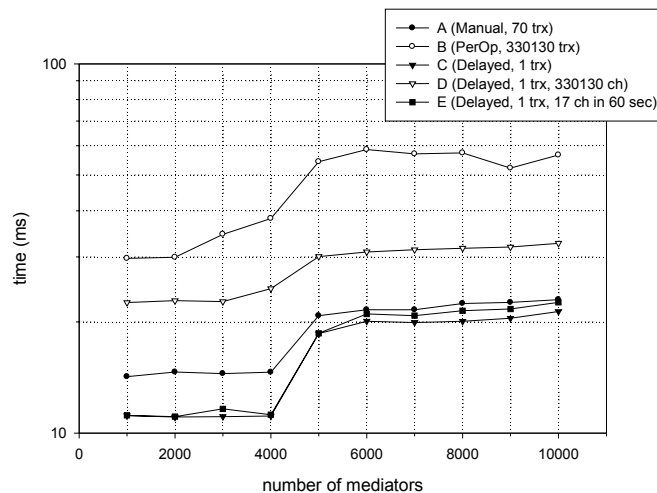


Figure 92. Total performance of all operations of the test application using different transaction styles

The presented results show that transaction handling is very costly and has significant influence on application performance. While the part of client-server communication remains constant, the times for method invocation on the client side differs drastically, depending on the transaction style set in the adapter. Frequent opening and closing of transactions proved to be very expensive showing performance overhead ranged from a factor of 2x to 4x. The natural conclusion is that CORBA/ODBMS applications can achieve much better performance by grouping multiple operations within a single transaction.

6.5 Designing robust CORBA/DBMS Applications

Although XDA makes it easy to create an integrated CORBA/ODBMS application in a short time, it does not guarantee the construction of a high-performance and reliable system. If you want to build something that scales to large numbers of objects and at the same time performs well, you must plan ahead and consider the underlying network infrastructure of the integrated system.

6.5.1 Design of IDL Interfaces

In CORBA-based systems, the IDL plays an important role in component interactions, as it defines interfaces at which servants comply. Hence, the first logical place for applying design techniques is in the IDL design.

Designing IDL interfaces is one of the most fundamental design activities and one in which major benefits of distributed programming can be gained and lost. It is not logically to expect to have a good distributed application simply „throwing together a handful of interfaces“ and then in some way implementing them. But what an interface will be ideal?

In general, an ideal interface

- presents a complete and coherent set of concepts to a user,
- is consistent over all parts of a component,
- does not reveal implementation details to a user,
- can be implemented in several ways,
- uses statically defined types (no Any or DynamicAny is used),
- is expressed using application-level types,
- depends in limited and well-defined ways on other interfaces.

Ideally, an argument of an interface function carries enough detail to make it self-describing. It should offer only operations that make sense and that can be implemented well by every derived class implementing that interface. However, this is not always easy. Consider lists, arrays, associative arrays, trees, etc. As it was shown in Section 5.5.6, it is sometimes useful to provide a generalisation of these types that can be used as the interface to every one of these. In our example an interface `Collection` is possible to use for all other types like `Set` or `List`.

However, defining an interface for a generic type is not trivial. What operations do we want to provide? We can provide only operations that every child interface can support. The intersection of the sets of operations for all interfaces is a ridiculously narrow interface. In fact, in many interesting cases that intersection is empty. Alternatively, it is possible to give a union of all the sets of operations and give a runtime error if a „non-existent“ operation is

applied to an object through this interface. An interface that is such a union of interfaces to a set of concepts is usually called a „fat“ interface or a *view* (see Section 4.3.3).

On the one hand, the technique of views solves the problems incurred by the *interface granularity* on large sets of small interfaces but, on the other hand, it introduces other problems connected with the performance and clearance of the code. Consider a „general container“ of objects of type `XDA_Collection`, which supports both the list-style operations and set-style operations as well. Doing so leads to the use of runtime type-inquiry or exception handling to avoid runtime errors. In both cases, runtime performance can suffer and the code generated from the IDL can be surprisingly large. Additionally, the use of fat interfaces weakens their correspondence to the concepts and thus opens the floodgates for their growth. As a result, fat interfaces are best avoided where runtime performance is at a premium and where strong guarantees about the correctness of code are required.

Consider a design that avoids inheritance. Common reasons for doing this are claims that „inheritance is an implementation detail“, „inheritance violates information hiding“, „inheritance makes cooperation with other software harder“. In many cases, there is no real advantage to be gained from inheritance apart of the ability of the client to cast references. Sometimes, however, it can be very useful. For example, in our case with operations on spatial objects it simplifies up/down casting of the references. Therefore, clean hierarchies of interfaces are not essential part of every good program, but in many cases they can help in both the understanding of the application and the expression of a solution.

6.5.2 Performance Issues

Another common design issue frequently overlooked is the difference between local and networked applications. The similarity of IDL and C++ class definitions makes it easy to think that IDL interface design is something similar to C++ class design. This similarity is both a boon and a bane. Because IDL is so similar to C++ in both syntax and semantics, most programmers quickly feel at home with the language and can begin to develop applications with only a small learning curve. However, pretending that IDL design is the same as C++ class design is definitely wrong.

Even though CORBA makes it easy to ignore details of distribution and networking, this does not mean that it is possible to pretend that distribution and networking do not exist. For example, it is easy to forget that sending a message to a remote object is several orders of magnitude slower than sending a message to a local object. For example, on an average UNIX workstation, it is possible to achieve easily more than 1000000 local C++ function calls per second, but only 200 or 2000 remote invocations per second. Naive IDL design, therefore, can easily result in systems that work but are unacceptably slow.

Therefore, from the technical perspective, making the transition from one to n-tier architecture requires risk management of issues such as network latency, system responsiveness, service availability, load management, distributed caching, distributed garbage collection, and system management. Fortunately, these issues can be mitigated if they are anticipated early in the design cycle. For example, if a system design seeks to minimise network traffic caused by interactions among distributed components, the system performance improves accordingly.

6.5.2.1 Reducing Messaging Overhead

The cost of request processing is determined by two main factors: *call latency* and *marshalling rate*. *Call latency* is the minimum cost of sending any request at all, whereas the *marshalling rate* determines the costs of converting parameters and return values by sending and receiving. Call latency also includes the costs of request dispatching. It varies considerably among

environments and depends on a large number of variables, such as the CPU speed, the operating system, and the efficiency of the dispatching implementation.

The cheapest message is one that has no parameters and does not return a result. For example, a `ping()` operation from our test application.

```
// IDL
interface XDA_Test {
    void ping();
};
```

The number of `ping()` invocations that XDA can deliver per time interval sets a fundamental design limit. To give a rough idea of the XDA's dispatching efficiency we made some benchmarks that show the average call dispatch times between 1.5 and 3 msec. In other words, using similar hardware and ORB implementation, it is reasonable to expect a maximum call rate of 20000 to 40000 operation invocations per second. See Section 6.3.7 for details about the test.

The following example shows a C++ class `gtTriangleNet` that models a 3D triangle net in `GeoToolKit`. This interface uses a fine-grained approach for creation of a new triangle net. All triangle objects should be created and inserted in the net individually with the help of the function `insert()`.

```
// C++
class gtTriangle;
class gtTriangleNet {
    ...
    void insert (gtTriangle *tr);
    ...
};

// IDL
interface GTA_Triangle;
interface GTA_TriangleNet {
    ...
    void insert (in GTA_Triangle tr);
    ...
};
```

The straightforward design of the corresponding IDL interface (presented in the right) is both clean and easy to understand, but it would require to perform a remote invocation for inserting every triangle. The construction of a new instance of `gtTriangleNet` automatically assumes the sequential invocation of two functions (`create_object()` and `insert()`) for all triangles of the net. To completely insert/retrieve the state of a net, the client must make many calls, one for each triangle. It is clear that on large triangle networks containing many hundreds thousands of triangles this design will suffer strong performance penalties.

One way to design a more efficient system, therefore, is to make fewer calls. For small parameters up to few hundred bytes the cost of a single invocation is essentially constant, so the programmer must try to make calls worthwhile by sending more data with each call. This design trade-off is also known as the „*bulk*“ operation technique.

Consider again the presented `gtTriangleNet` interface. Instead of modelling each triangle as a separate object, this interface provides the `insert()` operation to insert all triangles with a single call.

```
// IDL
interface GTA_Triangle;
typedef sequence <GTA_Triangle> GTA_TriangleNetRep;

interface GTA_TriangleNet {
    ...
    void insert (in GTA_TriangleNetRep tr);
    ...
};
```

The following Figure 93 compares performance of simple and „bulk“ implementations for an operation inserting the data of different size – 100*4 and 200*4 bytes. The results were obtained in the same configuration that was used for many other tests presented in this thesis. See details about this test application in Section 7.3. The client was running on a Sun SPARC Ultra 2 workstation (xenon) and the server on a Linux workstation (thror). ORBacus from IONA Tech. was used as an ORB implementation. The size of the ObjectStore’s client cache manager was set to 983040 bytes.

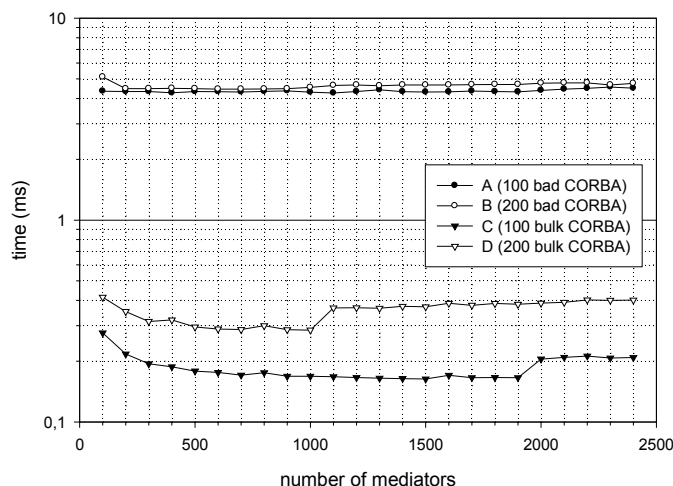


Figure 93. Performance comparison of „bad“ and „bulk“ implementations for `insert()` operation inserting the data of different size – 100*4 and 200*4 bytes

Here we see the benefit of the „bulk“ operations C and D that is about 4 msec per operation, the measured total time was divided by the total number of objects (invocations). The „jumps“ marked at 1000 and 1900 mediators can be explained by the caching effect, which happens when the total size of accessed objects exceeds the size of the cache used by the ObjectStore’s client manager.

In sum, since the difference in invoking an operation on a remote object (CORBA) is orders of magnitude slower than invoking an operation on the cached object (ODBMS), the system designer should do his best to prevent CORBA clients to invoke lightweight operations. The item-intense operations such as browsing a collection of fine-grained objects should be performed on the server side and only the result is transmitted to the client. This helps to reduce the amount of CORBA objects (and, consequently, objects references the client has to keep to the minimum) and prevent the communication overhead.

An alternative technique represents the realisation of the client’s cache that will allow to process fine-grained operations locally. Caching can improve performance and scalability of such applications by increasing the locality of data. An example of such an approach can be found in [WT00].

6.5.2.2 Improving the Marshalling Rate

The second factor that limits the speed of communication is the *marshalling rate* – that is, the time necessary to convert the data from the format used by a local application to the raw format that is used for data transmission. Marshalling performance depends on the type of the data transmitted. Simple types, such as arrays of octet, typically marshal fastest. On the other hand, marshalling highly structured data, such as nested user-defined types or object references, is usually much slower, because the adapter must do more work at runtime to collect the data from different memory locations and copy it into a transmit buffer.

For example, the following IDL code presents a commonly used representation of a triangle net by two sets – a set of points (`pointSeq`) and a set of triangles (`indexSeq`).

```
// IDL
typedef double GTA_PointRep[3];
typedef sequence <GTA_PointRep> GTA_PointRepSeq;

typedef unsigned long GTA_TriangleIRep[3];
typedef sequence <GTA_TriangleIRep> GTA_TriangleIRepSeq;

struct GTA_TriangleNetRep {
    GTA_PointRepSeq pointSeq; // points
    GTA_TriangleIRepSeq indexSeq; // indexes
};
```

The marshalling rate of the `GTA_TriangleNetRep` structure will depend on the representations of its components – `pointSeq` and `indexSeq`. The marshalling of highly structured data, such as nested user-defined types or object references, is usually much slower because the ORB and XDA's mediator must do more work at runtime to collect the data from different memory locations and copy it into a transmit buffer. Most ORBs also slow down significantly when marshalling any values containing complex data, mainly because the type codes for complex data are themselves highly structured.

To minimise marshalling times the both sets should be modelled by flat simple arrays as it is presented in the following code example. This is not surprising, considering that an ORB and the XDA can marshal such arrays by doing a simple memory block copy.

```
// IDL
typedef sequence <double> GTA_PointRepSeq;
typedef sequence <unsigned long> GTA_TriangleIRepSeq;
```

This approach is often called as *coarse types*. The main disadvantages of this optimisation are that it loses some type safety and does not support polymorphism. For example, the representation does not contain semantics about the correspondence of the values from the first array with the second (`pointSeq` and `indexSeq`). Each CORBA client must be explicitly aware of all possible types of representations that can occur, if more specialised versions of this representation are later added to the system.

In general, the coarse types approach works well, if objects are simple and their polymorphism is not required. In this case, objects provide set/get semantics for only a small number of attributes and so might as well be represented as structures. Coarse object models are similar to the bulk operation technique in that they reduce messaging overhead. However, the main value of coarse object models is that they can improve scalability, because they reduce the memory overhead for clients and servers dealing with large numbers of objects.

6.5.2.3 Transaction Management

The crucial role that the transaction management plays for performance of the integrated system was demonstrated on concrete examples as well as the effects of the new transaction management techniques. In particular, the efficiency of new methods for controlling transactions presented in this chapter was verified.

The XDA provides to the programmer a set of different transaction models: flat transaction, nested transactions, etc. Simply speaking all these models give different answers on two basic questions: when a transaction should be committed or aborted and when changes made by a transaction should become visible for other transactions. As far as different applications require different transaction models, the XDA left the answer on this question to the programmer.

However, the presented solutions do not exhaust all possible improvements that can be made. Particularly with regard to making small modifications to the runtime system, an opportunity for further optimisation could be a capability for detection of places, where iteration through arrays or collections of objects is performed. With small runtime modifications, we believe we can offer full transactional semantics (such as if manual transactions are used) for automatic transactions with reasonable performance.

6.5.3 Using Patterns to Build an Efficient CORBA/ODBMS System

The given recommendations for improving application's design can be summarised in the form of various design patterns. The intent and usage of some of them are outlined below.

- **Facade** [GHJ+94] or **Framework pattern** [MM97]. Its intent is to encapsulate fine-grained interfaces within modular large-grained interfaces defining the external IDL interface of the application. Thus, the facade pattern creates a higher level of abstraction between underlying database and CORBA environment.

For example, the following C++ classes can be represented by only one IDL interface to be exposed to CORBA clients. Unfortunately, it is not always possible to separate two kinds of interfaces completely. For example, an operation defined in a large-grained interface may take as an argument or return the value of a variable described by a fine-grained interface.

```
// C++
class Date {...};
class Address {...};
class Person {
    char* name;
    Date& date_of_birth;
    long id;
    boolean gender;
    Address* adr;
    ...
};

class Population {
    collection<Person*> pop;
public:
    long population_number(char* city_name);
    short average_age(char* city_name, boolean gender);
    char* most_popular_name(char* city_name, boolean gender);
    ...
};
```

```
// IDL
interface Population {
    long population_number(in string city_name);
    short average_age(in string city_name, in boolean gender);
    string most_popular_name(in string city_name, in boolean
gender);
    ...
};
```

- **Collection interfaces.** In cases, when in the database a collection of objects of the same type is stored, it is possible to define one interface carrying all collection-specific operations, like querying the content of the collection, getting the members of the collection, invoking operations on the subset of the collection (resembles „bulk“ operation technique) etc. For example, an interface like this one.

```
// IDL
interface Foo {
    void op();
};

typedef sequence<Foo> Foo_sequence;

interface Foo_collection {
    Foo_sequence get_all();
    Foo_sequence query(in string query_string);

    void op(in Foo_sequence seq);
    void op_on_query_result(in string query_string);
};
```

- **„Bulk“ operations.** This pattern groups several methods invocation into one „bulk“ operation with the arguments/results chunked in a structure. For example, operations A_op, B_op and C_op on the interface Foo can be gathered together and represented as one „bulk“ operation ABC_op.

```
// IDL
interface Foo {
    result_a_type A_op(in arg_a_type);
    result_b_type B_op(in arg_b_type);
    result_c_type C_op(in arg_c_type);
};

// IDL
struct arguments {
    arg_a_type arg_a;
    arg_b_type arg_b;
    arg_c_type arg_c;
};

struct results {
    result_a_type result_a;
    result_b_type result_b;
    result_c_type result_c;
};

interface Foo {
    results ABC_op(in arguments);
};
```

- **Coarse types.** Its intent is to decrease marshalling times by representing implementation-specific types with primitive CORBA types. For example, considering a database class `Date`

```
// C++
class Date {
private:
    // time, day, year, etc.
    ...
public:
    // constructors, destructor
    // type conversion operators
    // e.g. operator const char *();
    // member functions
    // additive operators (+, -, ...)
    // assignment operators (=, +=, -=, ...)
    // relational operators (<, >, >=, <=, ...)
    // equality operators (==, !=, ...)
};
```

A naive straightforward representation of this class in the IDL is to use an interface. The objects of this class are typically used in other interfaces such as `Bank` and can have an interface like this.

```
// IDL
interface Date {...};
typedef sequence<Date> Dates;

interface Bank {
    readonly attribute Date foundation_date;
    Dates get_modification_times(in long);
    ...
};
```

However, it is more pertinent to simplify the interface and, instead of exposing all methods/attributes of the `Date` class by defining new interface, replace the complex `Date` type by a simple built-in type, for instance, `string` type.

```
// IDL
typedef sequence<string> Dates;

interface Bank {
    readonly attribute string foundation_date;
    Dates get_modification_times(in long);
    ...
};
```

This technique reduces marshalling times for `Date` values but, in the same time it loses semantic for internal `Date` components such as time, date, year, etc. All in this case necessary conversions between a such simplified and an original complex representation of `Date` should be realised within corresponding mediator classes (e.g. by realising a type conversion operator for the class `Date`).

Chapter 7

Case Study: A distributed 3D/4D GIS

In the previous chapters the XDA's architecture has been presented and its implementation issues have been discussed. Besides, the operating of the CORBA/ODBMS system, which has been integrated through an adapter developed on top of the XDA and the methods of its performance improvement have been reviewed. The present chapter illustrates how the XDA can be applied in real-life situations and which practical benefits it provides.

These issues are discussed on a prototype of the distributed database-supported GIS considering the selection, manipulation and visualisation of 3D/4D geological data. The prototype was developed to support the 3D/4D modelling process in the SFB350⁴² C4 research project and was tested with an application area, the Bergheim kinematic model situated in the Lower Rhine Basin [Koo02, STCK02]. The developed prototype has a distributed open CORBA-based architecture. It consists of a portable Java-based client and object-oriented spatial database components connected to CORBA using a proprietary GeoToolKit/CORBA Adapter (GTA) [SB00, SS01], which was built with the help of the XDA development framework. Implementation aspects and open problems are discussed. In addition, the influence of the mediator and transaction management on the performance of the integrated CORBA/ODBMS system is studied.

The organisation of this chapter is the following: first, a brief motivation for the development of a distributed GIS will be given and some examples of complex geological models are presented. These models, together with the necessary geological background, are described in Section 7.1.1. The architecture of the prototype have developed is presented in Section 7.2. It describes all basic components and lists the key technologies used. Thereafter the steps performed to build the integrated CORBA/ODBMS application are depicted and the results of the application of the XDA are briefly summarised. The specific profits for the application by wrapping persistent objects stored in the input database are briefly described. A more detailed discussion of techniques that allowed us to build an efficient distributed system follows in Sections 7.2.2, 7.2.3 and 7.2.4. The techniques for database navigation, visualisation and progressive transmission are discussed. We conclude with a performance evaluation and a report of the results achieved.

7.1 Motivation

The fast growing number of spatio-temporal applications in fields such as environmental monitoring, geology and mobile communications presents new challenges for the development of geo-information systems. In contrast to classical 2D GIS dealing with the Earth Surface, no typical or standard way of managing and analysing subsurface 3D and 4D geometry data has yet evolved. On the one hand, the introduction of two additional dimensions causes an enormous increase in the volume of raw data resulting in the need for its compact storage and transmission. On the other hand, the analysis of geo-scientific data by means of different problem-specific tools requires the use of the interactive query processing facilities of the database ranging from complex set-based operations such as spatio-temporal joins to simple selections for on-line visualisation [BCG+99, BBCS00b,

⁴² Collaborative Research Center (SFB350) „Interactions between and Modelling of Continental Geo-Processes“.

BBCS00c, BS01]. Therefore, new techniques are required which can provide a reasonable compromise between compact storage and efficient retrieval of spatio-temporal objects.

7.1.1 Examples of geological 3D/4D Models

Within the frame of the long-term collaborative research project SFB350 at Bonn University, a group of geologists, lead by A. Siehl, developed a number of 3D geometric and kinematic models of selected regions within the Lower Rhine Basin. These models, differing in extension, scale and detail, are a general, kinematic, inclined-shear model of the Erft block [ABB+98], a subsidence and compaction model of the Rur and Erft Block [JS02], and a kinematic model of the Bergheim area [TS02] comprised time-dependent geometry objects (Figure 94).

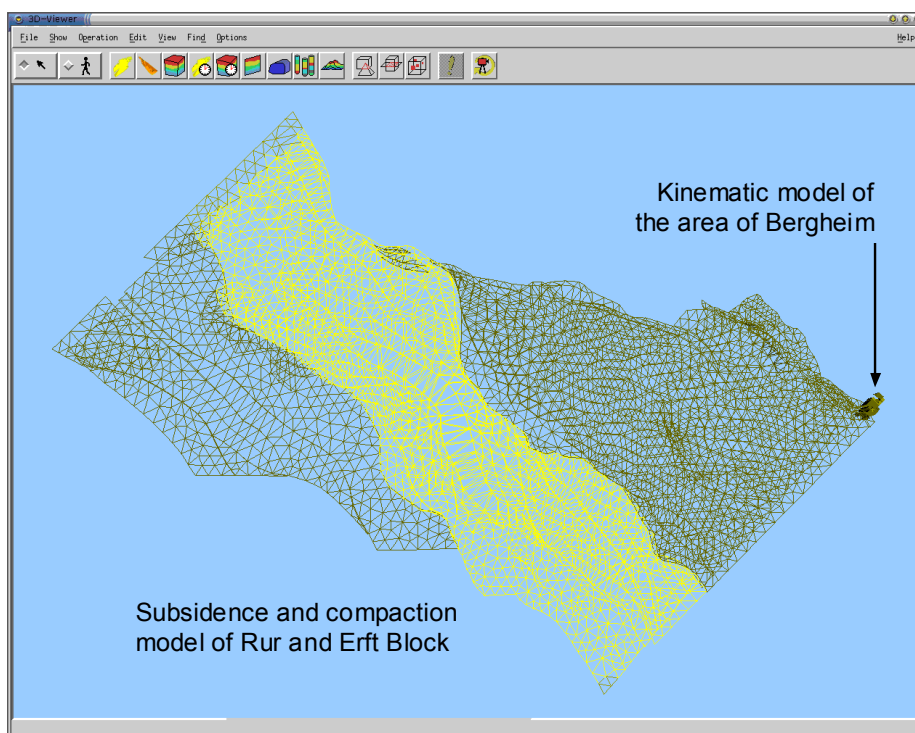


Figure 94. Two geological models in the 3D viewer of GeoToolKit

To gain a deeper insight into the development of the Basin, the kinematic model of the small and intensively faulted region around the Bergheim lignite mine, with an extent of about 2 km and a maximal depth of 500 m, was developed. The model comprises about 14 stratum layers, disrupted by more than 100 faults. Whereas earlier models of the Lower Rhine Basin, having lateral extensions of several tens of kilometres, were modelled using free-form surfaces, the Bergheim model was designed as a structure of several hundred volume blocks, to support relative displacements at fault surfaces and small deformations.

Figure 94 shows the structure of the Bergheim geometry model. The geometry at a given time step is represented using triangulated surfaces and complex fault-delimited volume blocks with a boundary representation. The blocks of the Bergheim model combine three types of triangulated surfaces – namely stratigraphic boundaries, faults and artificial boundaries – that follow the original geological profile sections. Their positions are marked by hatched lines at the top of the blocks. Figure 95 shows the positions of stratigraphic boundaries and places of possible displacements at faults by arrows.

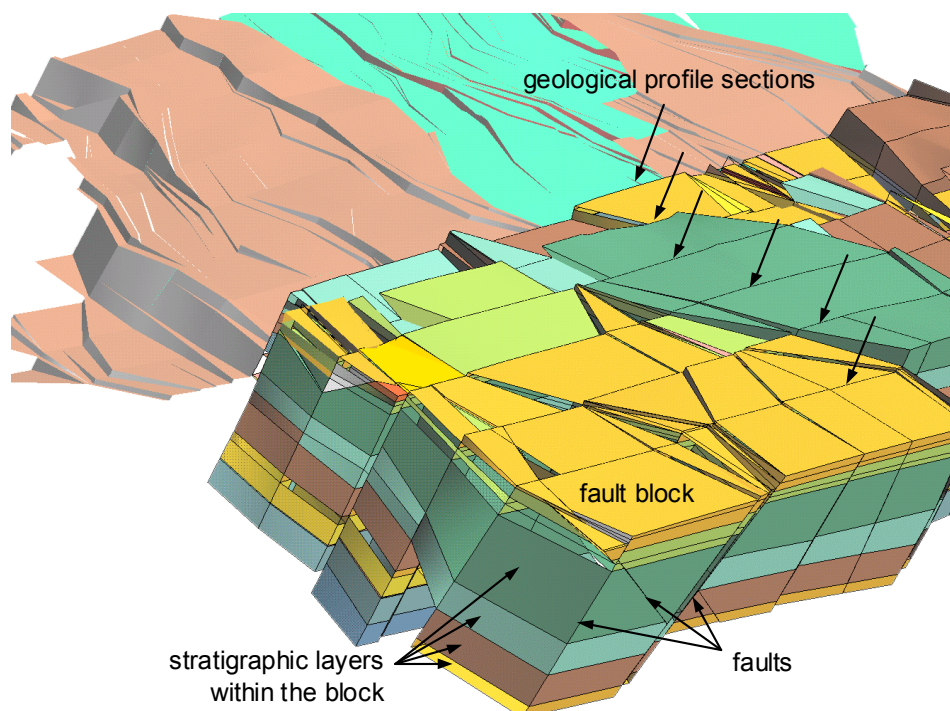


Figure 95. Structure of the Bergheim geometry model

The interactive design of the geometry was performed with the GOCAD[™] [Mall92, Tsurf02] 3D modelling program, while the GRAPE [Jam99] library was used for the visualisation and animation. Time dependency is essentially realised by triangle meshes and vertex coordinates changing in time. Rouby's method of map plane restoration [RSB+96] was applied to individual horizons to determine reconstructed states within a time interval from about 18 million years ago to the present day [JS02]. During block movements, small inconsistencies at block borders are accepted and taken into account by Rouby's reconstruction method.

7.1.2 Requirements for the Storage and Retrieval of spatio-temporal Objects

As geo-scientific information on the subsurface generally is too sparse to support an automatic generation of useful 3D models, the construction of geological 3D geometry models of the underground must rely not only on information obtained from the analysis of hard data gained from outcrops, boreholes, seismic profiles etc., but also on a considerable amount of geologist's interpretation and hypotheses guided by geo-scientific knowledge, and controlled by consistency and plausibility considerations. Both kinds of information often represent considerable investments. Acquiring raw data and its analysis involves costly technical equipment and skilled manpower. The construction of a 3D geometry model by geoscientists using sophisticated interactive geometry modelling software frequently consumes a large amount of time. A geological analysis also involves an interpretation of the development history of the present situation, e.g. by kinematic analysis of basin development, resulting in an increased interest in spatio-temporal models.

Whereas the Bergheim kinematic model consists of a large number of complex volume blocks separated by simple boundary surfaces, in many other cases geological 3D models comprise dozens to hundreds of triangulated surfaces, sometimes consisting of about a hundred thousand triangles or more. The size of the model is multiplied by the addition of the time dimension. The tools are required which can provide a reasonable compromise between compact storage and efficient retrieval of spatio-temporal objects. Most modern 3D

modelling tools do not have these features and employing them directly for the construction of complex large spatial models is not possible. Therefore, an approach for coupling heterogeneous 3D modelling tools with spatial databases was proposed [STCK02]. The new technique provides a reasonable compromise between compact storage and efficient retrieval of spatio-temporal objects.

7.2 CORBA-based integrated Architecture

During our long-term collaboration with the geologists, we identified the need for a flexible distributed infrastructure capable of integrating the existing heterogeneous and highly-specialised geoscientific tools with spatial database management systems. The Base Communication Infrastructure that we developed provides a set of object services and specific objects for the management of persistent interactive user sessions which make use of data servers encapsulating data from a variety of sources [BBCS00a, BRSC02]. It also defines communication interfaces and protocols that perform as a common interaction „language“ between all components of the system. Using standard middleware technologies, the infrastructure is open and independent of any specific application domain. It is designed to support the full range of data services and client applications across the entire application domain and beyond. Based on this infrastructure, a prototype of a distributed 3D/4D GIS was developed to support the construction of complex geological kinematic models (Figure 96). It consists of the components noted in the following sections.

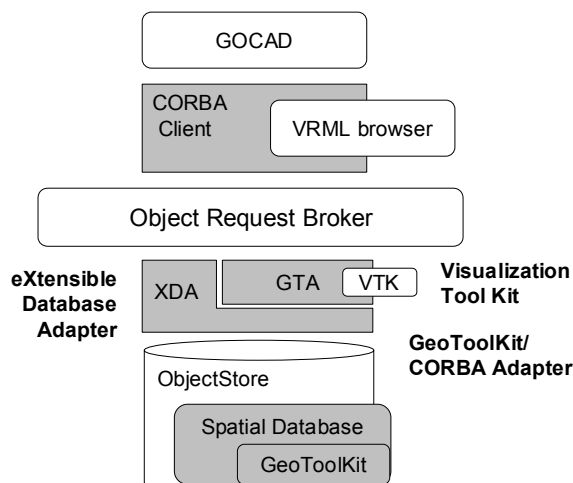


Figure 96. CORBA-based integrated architecture of the prototype

7.2.1 Spatio-temporal object-oriented Database

An object-oriented 3D/4D database developed on top of GeoToolKit [BBC97] provides the management of persistent spatio-temporal objects. The GeoToolKit library is an object-oriented, spatio-temporal 3D database kernel realised on top of the ODBMS ObjectStore [ODIug01]. GeoToolKit gives spatial database developers a set of geo-oriented building blocks incorporating efficient methods for ODBMS-based spatial data maintenance and clustering. Currently, GeoToolKit offers classes for the representation and manipulation of simple (point, segment, triangle, tetrahedron) and complex (curve, surface, solid) 3D spatial objects. An additional layer of spatio-temporal (4D) classes enables the modelling of spatial objects which change their location and shape in time [SS01].

7.2.2 GeoToolKit/CORBA Adapter

The DBMS is made available to the Common Object Request Broker Architecture (CORBA) environment with the help of a special adapter – GeoToolKit/CORBA Adapter (GTA), which was developed on top of an eXtensible Database Adapter (XDA) – our prototype of a CORBA/ObjectStore object adapter development framework [SC00]. It provides the necessary primitives and basic mechanisms to define, manipulate, and share persistent data in CORBA. The GTA represents an extension of the XDA by GeoToolKit-specific classes that are necessary for remote interaction with persistent GeoToolKit objects from the CORBA environment. Covered by the GTA adapter, GeoToolKit can be regarded as a distributed 3D/4D geo-database made available for remote CORBA-compatible clients.

To speed up both the transmission and visualisation of 3D objects on the client side, support for progressive spatial data transmission has been implemented in GTA. It reduces the size of the original spatial data by using a special format for its progressive transmission through the CORBA. The algorithm for progressive mesh decimation on the server side was implemented with the help of a freely available computer graphics library *Visualisation Tool Kit (VTK)* [SML97].

7.2.3 Portable Client Interface

The integration of geo-scientific applications with the CORBA-based infrastructure was achieved by developing a problem-specific, portable, client interface. The platform-independence of the interface is supported by the Java execution environment [SUNj02]. The client communicates with remote GeoToolKit-based databases by CORBA's IIOP protocol and provides querying, previewing and conversion of retrieved objects into application-specific formats (Figure 97).

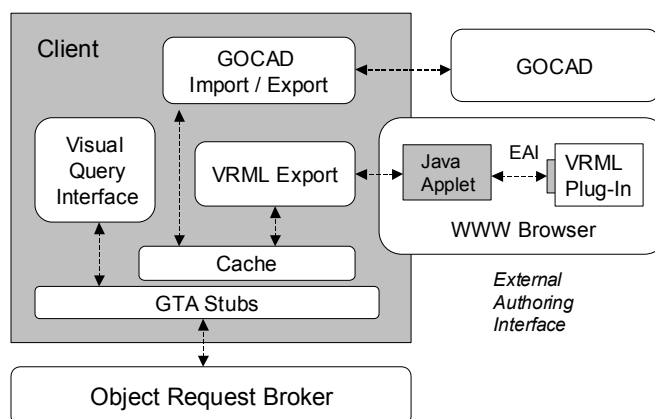


Figure 97. Architecture of the client and extension of a VRML-browser for visualisation of time-dependent spatial data

Retrieved objects are kept in a cache for the reuse in subsequent queries or for forwarding to geo-scientific applications for further treatment. Currently the client supports GOCAD and VRML output formats [VRML97]. The data retrieved can then be visualised using a VRML browser before they are stored or forwarded to a geoscientific application for further processing. By means of a special extension (Java applet), it is possible to observe and control the movement of selected objects in time. Considering the size of complex spatio-temporal geoscientific models, the client can take full advantage of progressive data transmission when previewing large models at a lower level of detail while avoiding redundancy when higher resolution is required. A more detailed discussion of database navigation, visualisation and progressive transmission techniques follows in the next sections.

7.2.3.1 Database Navigation and Queries

A typical session with the client interface might proceed as follows. During work with a 3D/4D modelling application, the user wishes to retrieve geometry objects from a remote database for insertion into his local model. For example, to select volume blocks from a distinct region. After starting the client interface and establishing a connection with the database, a GUI window with several sections appears (Figure 98). Its listbox windows display possible attribute values, (section A from right to left) which can be used for object selection. Objects and their parts (attributes) are classified according to three basic geological types: „Layers“ (stratigraphic surfaces), „Faults“ (tectonic discontinuities), and „Profiles“ (geological sections). The list „Blocks“ displays a fourth type, the volume blocks of the model. These geological types are typical for many geological applications, and do not limit the usability of the interface to the single Bergheim model. The interface helps the client to navigate through the database following the attributes of selected objects. For example, in Figure 98, an attribute „Profile“ with value B231 has been selected. This selects all objects having the „profile“ boundary surface B231 as an attribute. The selected profile value is marked in dark grey. In addition, we observe that a number of „Fault“ and „Layer“ values are marked in light grey, signalling the other attribute values taken by the selected objects.

The retrieval of selected objects from the database is performed with the help of queries (C), which are activated with a button „Export“ (D). Objects selected for retrieval in one of the listboxes (Figure 98, A) can be inserted in queries using one of the buttons „Intersection“, „Union“ or „Not“ (B). The user may modify the selection condition by choosing additional objects in the same or other listboxes. Additionally, by activating button „Details“ (D) it is possible to investigate the internal structure of retrieved objects or to select only their parts for retrieval.

Complex Queries

To define more complex queries, the user may use a query constructor and manipulate the query tree displayed in the lower part of the GUI (C). The query is composed of boolean operations (union, intersection or not) combining simple type-value pair conditions. Figure 98 displays a query consisting of an intersection of two union operations: the first operation selects objects with the attribute „Profile“ values B237, B231 and the second selects objects having at least one of the displayed „Fault“ values. The user can add new operations to the query by activating corresponding buttons at the right side of the window (C). For example, activating button „Add Condition“, the user can define a new condition and combine it with the selected boolean operation. Other buttons permit changing the type of an existing operation and removing either a subtree of the query or the whole query.

In this particular prototype, we did not implement the purely spatial, e.g. bounding box, queries provided by GeoTooKit. In the present application, attribute-based queries provide sufficient problem-specific criteria and navigation facilities. However, spatial queries may be added to the client in the future.

Time-dependent Queries

When retrieving time-dependent geometry objects, the user may specify a time instance t or a time interval $[t_1, t_2]$ by means of two slide rulers at the right side of the query window (Figure 98, E). If a single time instance t is specified, a 3D „snapshot“ of the object at time t will be returned. If instead a time interval $[t_1, t_2]$ is specified, a new 4D object resulting from truncation of the original 4D object is returned with $[t_1, t_2]$ as the new interval of validity. If necessary, the object's states at the boundaries will be interpolated. Obviously t and t_1, t_2 must be inside the original interval of validity. By means of a VRML browser extended by a Java applet, it is possible to observe and control the movement of selected objects in time (Figure 99).

Queries at multiple Resolutions

The volume block objects of the Bergheim model consist of a limited number of surfaces with few triangles. This is not typical for geoscientific applications, where often several dozens of large triangulated surfaces with up to some 100000 triangles must be managed. As an example, a test set of 3D stratigraphic surfaces of the Lower Rhine Basin was investigated (Section 7.2.4.2).

To reduce the considerable amount of time spent on data transmission and visualisation in such models, methods for progressive transmission [Hop96] were studied and adapted for geoscientific applications. When working with large scale models, it is reasonable first to retrieve queried objects for an overview of a large area at reduced resolution, and later to add detail only to those objects that are finally requested. Therefore, a second group of slide rulers (Figure 98, F) allows the user to specify a reduction factor ranging from 0.0 (no reduction) to 1.0 (a maximal resolution determined by constraints imposed on the decimation). Using this technique, it is possible to significantly reduce the time for retrieval and for the rendering of large models.

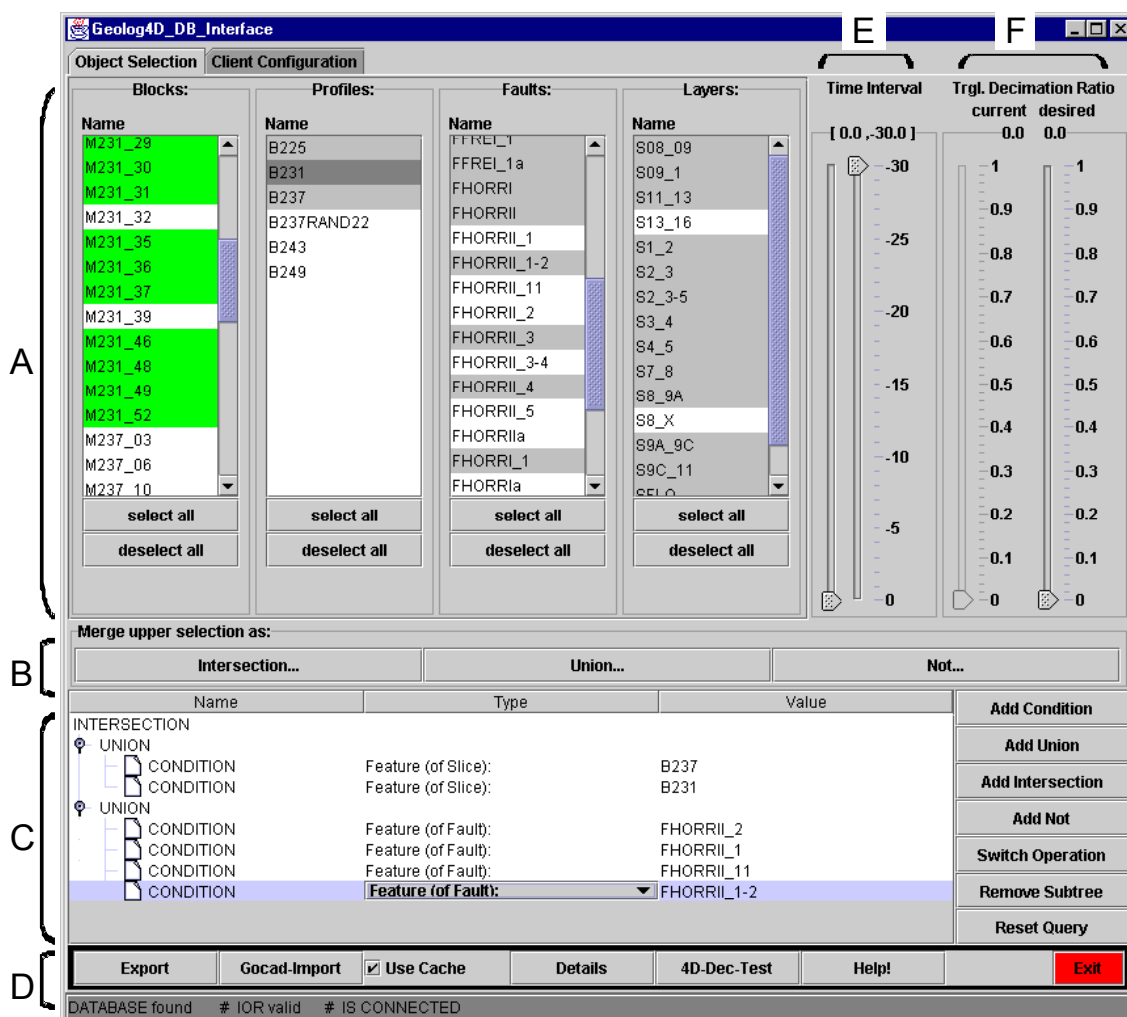


Figure 98: A screenshot of the client’s graphical user interface

7.2.3.2 Integration with Tools for 3D/4D Visualization

A preview operation will be often used in practical work before loading the selected data for further processing in specialised geoscientific applications. Basic geological 3D entities turned out to be well mapped on to the VRML representation [BBBS99]. Therefore, when a

graphical representation is required, the retrieved objects are converted to VRML and visualised using a common VRML-capable WWW Browser. However, VRML offers much more advanced facilities, such as the animation of 4D objects (Figure 99). The animation is defined by a set of keyframes representing 3D geometries for critical moments in an animation cycle. *CoordinateInterpolator* nodes are used to compute all the intermediate positions between keyframes that are necessary for a smooth animation. Whereas well known VRML browsers support animation, they do not provide sufficient control of animations as they evolve. For this purpose, we have studied two different mechanisms: time sensors and an additional Java applet.

TimeSensor nodes provide a simple and natural way to control an animation in a VRML browser. Floating point values between 0 and 1, generated by the sensor, are routed to coordinate interpolators, which use these values as keys to calculate the interpolated values. The duration of animation in this case is equal to the value of the *cycleInterval* of the corresponding time sensor. An animation of a group of visualised geological objects is started by using the *TouchSensor* node defined in the group and routing the events from the touch sensor to the time sensor. The touch sensor detects mouse clicks anywhere in the group and generates an event that starts the time sensor. However, since touch sensors can only start and stop the animation, this functionality is still insufficient. To control the consistency of complex kinematic modes it is necessary to start and stop the animation at user-defined times. The mechanism of time sensors is unusable in this case because there is no way to limit the interval of key floating point values generated within the cycle.

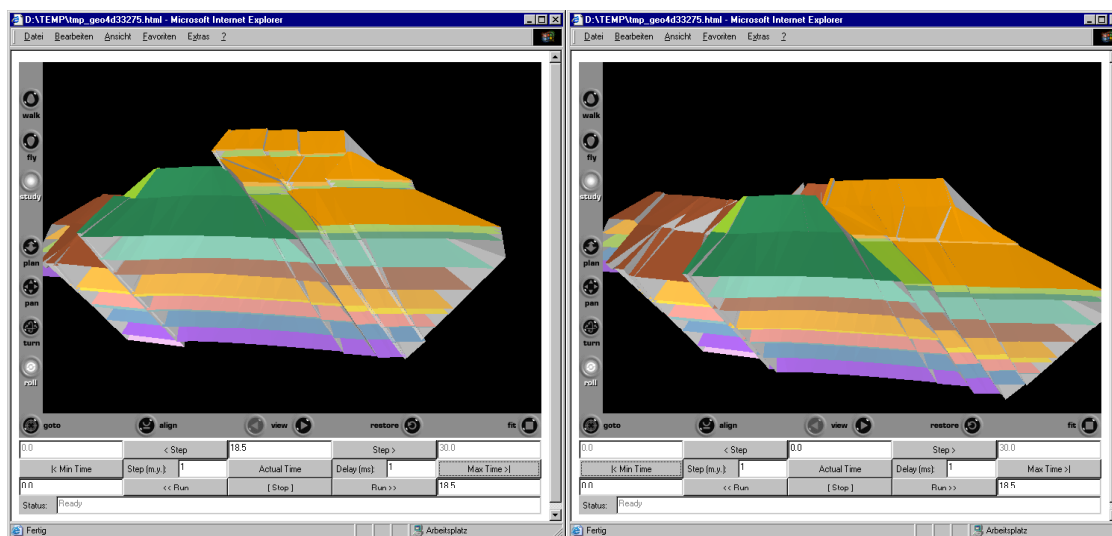


Figure 99. 4D geometry objects at different time steps -
above: 18.5 million years ago, below: today -
in a VRML browser (in our prototype - Cortona™ [Par02])

Therefore, another method is developed to control the visualisation process in the second version of our prototype. The original *TimeSensor*'s functionality was emulated by means of a Java applet generating key floating point values and a simple VRML Script forwarding these values to *CoordinateInterpolator* nodes. The applet communicates with the VRML-browser through the *External Authoring Interface* (EAI) and provides a simple user interface that allows the user to control the animation. Interacting with the applet the user can explore the model at particular time steps, start and stop an animation at arbitrarily times, change its direction and time interval boundaries. Figure 99 shows the interface in the window of VRML browser with several tectonic blocks from the Bergheim kinematic model at two time steps.

Employing the automatic generation of VRML objects with the powerful combination of VRML scripts and Java, it was possible to achieve exact control over the multiple synchronised animations, e.g. several blocks moving with different directions and speeds. Actually, using this method, it is also possible to create more sophisticated interpolators than VRML97's built-in linear interpolators, and use Bezier, B-spline, and cardinal curve methods to obtain smooth interpolation with fewer keyframes.

7.2.4 Transmission of large and complex Objects

In object-oriented client-server systems the overhead required for data transmission from server to client and the control of such transmissions is significant. As users of the prototype have noticed, fetching and updating data from many small objects kept at a remote object store is costly. In fact, high-latency and low-bandwidth transport networks, such as the Internet, require a reduction in network traffic to improve the overall performance of the system. These limitations of the transport network have led to a different paradigm in distributed processing systems called progressive data transmission.

7.2.4.1 Standard Representation for transmitted Data

Simply looking at the database schema often it is not clear what method should be used for a representation of references. Usually it is necessary to define several kinds of representations describing the value of reference in several different ways. For example, using the standard struct-based IDL mapping for values defined in Chapter 4, the value of a triangle can be represented by the following two structures: `GTA_TriangleRep` and `GTA_TrianglePRep`. The both structures defines the value as an ordered set of 3 points. But the first one (`GTA_TriangleRep`) defines points as values and the second (`GTA_TrianglePRep`) as references.

```
// IDL
interface GTA_Point;
interface GTA_Triangle;

// Point
typedef double GTA_PointRep[3]; // representation by value
typedef sequence <GTA_PointRep> GTA_PointRepSeq; // sequence of values
typedef sequence <GTA_Point> GTA_PointSeq; // sequence of references

// Triangle
typedef GTA_PointRep GTA_TriangleRep[3]; // representation by values
typedef GTA_Point GTA_TrianglePRep[3]; // representation by references
```

7.2.4.2 Mesh Generalisation and Progressive Transmission

Real 3D data sets and especially time-dependent 3D data sets are usually large. In this case the preview operation is not always really useful because of the intolerable slow performance. Our *GeoWeb* study [BBBS99] showed that this was rather due to the slow performance of the network and visualization, than to the applied software and algorithms. Some experiments with *GeoWeb* made clear that the problem can only be solved by reducing the complexity of the visualised data. In most cases where large 3D/4D objects are used, a coarse representation of selected objects is enough for their unambiguous identification.

The standard facilities of VRML are not enough for manipulation with large data sets. The *Level Of Detail* (LOD) node is typically used in VRML for switching between different levels of details at specified distances from the viewer. On the one hand, this feature is useless for really large data sets. Requiring that geometries for all resolutions are already present in

VRML input, usage of LOD nodes follows to enormous amount of data transfers and amazingly slow rendering. On the other hand, it is not possible to find a general definition of how many details can be skipped. It rather depends on the particular kind of data, its semantic and its interpretation. The approximation of a required LOD could only be performed by the development of algorithms specific for every application domain. Therefore, in the general case, it seems more realistic not to try to find the optimal LOD, but to provide a possibility to varying the LOD for the data to be transmitted starting from a minimal coarse resolution and going in the details, if necessary.

The idea of progressive mesh generalisation is to process a mesh through the inverse of the decimation process. First, the mesh in the coarsest resolution is encoded, followed by a sequence of refinement operations, which are the inverse of the simplification operations. The inverse of vertex elimination is vertex insertion and the inverse of the edge collapse operation is the vertex split operation. This representation is ideally adapted for streaming meshes, as a mesh can be viewed in a coarse resolution before all vertex split operations have been transmitted. This feature is also known as *progressive transmission*. Hoppe [Hop96] was the first to come up with the progressive mesh representation – which is based on vertex split operations, followed by other more complex methods [BPZ99, PR00].

In our prototype a bi-directional progressive representation for 3D meshes in CORBA was developed. It allows for a remote CORBA client to gradually improve the resolution, according to the requirements of a concrete model and application. Through the sequence of edge collapse operations the geometry to be transmitted is converted into a progressive generalised mesh form, which is represented by a minimal mesh in a coarse resolution following by a sequence of refinement operations – deltas (Figure 100). The specified maximum approximation error defines the maximal level of decimation that can be achieved. Upon receiving a client’s request, first only the minimal mesh is transmitted to the client and used for visualisation in VRML. If the user decides that this resolution is not enough, he/she can gradually refine the mesh by requesting the necessary deltas from the server.

The functionality for progressive mesh decimation was implemented as a part of the *GeoToolKit/CORBA Adapter* (GTA). Implementation of a basic edge collapsing operation was taken from a freely available computer graphics library – the *Visualisation Tool Kit* (VTK) [SML97]. More details about the developed method can be found in [STCK02].

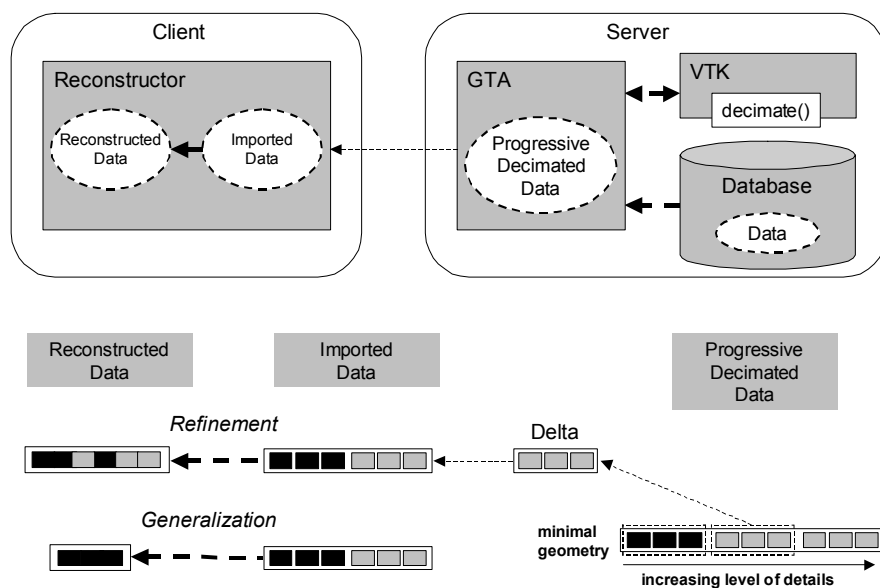


Figure 100. Architecture for progressive transmission

7.3 Performance Evaluation

As described above, the proposed persistent system is built by integrating the middleware with the database management system. Such systems will succeed or fail as much on their performance as a middleware as on their performance as a ODBMS. Thus, benchmarking such a hybrid system requires a holistic approach that focuses on both middleware and runtime behaviour as well as on the underlying database functionality. Performance metrics should include fine-grained (i.e., request dispatching) overheads in addition to the coarse-grained features typically measured by database benchmarks (e.g., transaction throughput, commit latency, disk accesses, etc.) [Hugh97].

Several attempts for benchmarking ODBMSs exists. For example, the *Simple Database Operations Benchmark* [RKC87], *Altair Complex-Object Benchmark* [DFMV90], *OO1* [Cat91, Hosk95], *OO7* [CDK+94, CDN93] and other [Dan99]. Benchmarks were created to provide useful insight for end-users evaluating performance of ODBMSs. However, none of the currently existing benchmarks were designed to adequately exploit the distinctive features of the hybrid system. Unfortunately, most benchmarks are highly specialised on particular systems for which these tests were developed [RVV+00]. They enforce a specific implementation that cannot be efficiently translated to different systems. The similar situation we have with middleware's benchmarks. The majority of existing CORBA benchmarks is designed to evaluate features native to CORBA and none of them are suitable to evaluate the performance for the features characteristic of database applications [Bubl99, TuBu01]. Therefore, a simple ad-hoc test application was designed.

7.3.1 Benchmark Description

A key component of the test is a set of persistent composite parts. Each composite part corresponds to a design primitive such as a spatial object in GIS. The set of all composite parts forms what we refer to as the „test database“. Composite parts are implemented by persistent instances of a class `Part`. The IDL interface of the corresponding mediator class `XDA_Part` is presented in the following Figure 101.

```
// IDL
typedef sequence <unsigned long> XDA_LongSeq;
typedef sequence <XDA_LongSeq> XDA_DataRepSeq;

interface XDA_Part : XDA_Object {

    attribute long id;
    attribute XDA_List children; // XDA_Part objects
    attribute XDA_LongSeq data;

    void ping();

    void create_children
        (in unsigned long number, in unsigned long size);

    void set_data (in XDA_DataRepSeq dataRep);

    void multiply_read_good (inout XDA_LongSeq data);
    void multiply_write_good (inout XDA_LongSeq data);

    void multiply_read_bad (inout XDA_LongSeq data);
    void multiply_write_bad (inout XDA_LongSeq data);

};
```

Figure 101. IDL interface of the test application

The number of composite parts per module in the test database is controlled by the parameter `children`, which is implemented using a list `XDA_List`. It is assumed that one composite part will be used as a database root to access other objects. Each composite part has a number of attributes, including the integer attributes `id` and `data`. The array of integer data is connected with the composite part object through a „soft“ reference.

In addition to its scalar attributes each composite part has several operations. Operation `ping()` does nothing. It is used for measuring performance of the XDA's dispatching. Results of this test are presented in Section 6.3.

The next two operations `create_children()` and `set_data()` are „bulk“ operations, i.e. operations that are composed of several simple operations. The first operation `create_children()` creates a (number) of children objects with data of a given (size). The second `set_data()` is used to set the array of integer data new values.

The last four operations simulate data-intensive operations, which are typical for a wide range of applications, for example geo-information systems GIS or digital signal processing DSP applications. The operations multiply incoming array of integer (data) with similar arrays in all children objects. The operations differ by the type of access to the test database (read/write) and by the implementation (good/bad). Figure 102 illustrates the difference between implementations of „good“ and „bad“ operations.

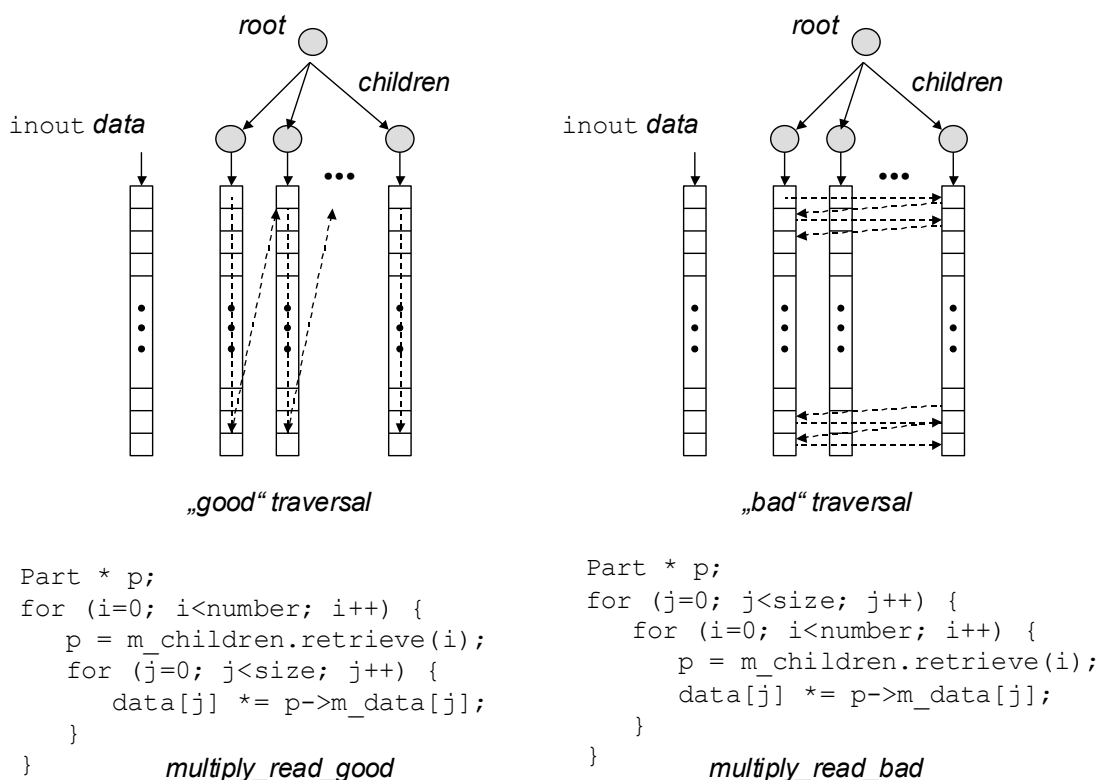


Figure 102. Navigation schemas of „good“ and „bad“ multiplication operations

The presented „good“ and „bad“ operations are designed to test the efficiency of clustering and client-side caching mechanisms of the ODBMS, which affect significantly the total performance of the ODBMS and, as result, the integrated ODBMS/CORBA system. Both types of operations perform the same mathematical operation for multiplication of all elements in the incoming array with similar arrays in the children objects. The difference between them is only in the way how the operations are navigating through children objects.

The „good“ operations frequently take objects located next to each other and, therefore, have a much more better performance than „bad“ operations frequently jumping between distant objects. Thus, performing „good“ operations the database can avoid frequent retrieval of remote data, keeping objects in the local client’s memory to avoid another retrieval if they are needed again later. On the „bad“ operations this strategy does not work and results in the worse performance than in the first case. In the both cases the performance of the system on update operations gets even worse than on read-only operations due to an additional communication overhead to insure the consistency of data after updates. All this only confirms the thesis that client-side caching is a major factor why the performance of ODBMSs depends much more on the client’s hardware than the performance of RDBMSs.

7.3.2 Test Environment

The test consists of a sequence of the above defined operations: `create_children`, `set_id`, `set_id`, `set_data`, `multiply_read_good`, `multiply_write_good`, `multiply_read_bad`, `multiply_write_bad`, `delete`, which was performed on variable numbers of persistent objects. All operations are performed within separate manual transactions. Simple operations iterating through a set of objects such as `set_id` were grouped together and performed within one common transaction.

To evaluate the performance of our CORBA-based system the test was performed for three configurations that are presented in Figure 103. It is a remote ObjectStore test application – A, local ObjectStore test application – B and remote CORBA-based test application – C.

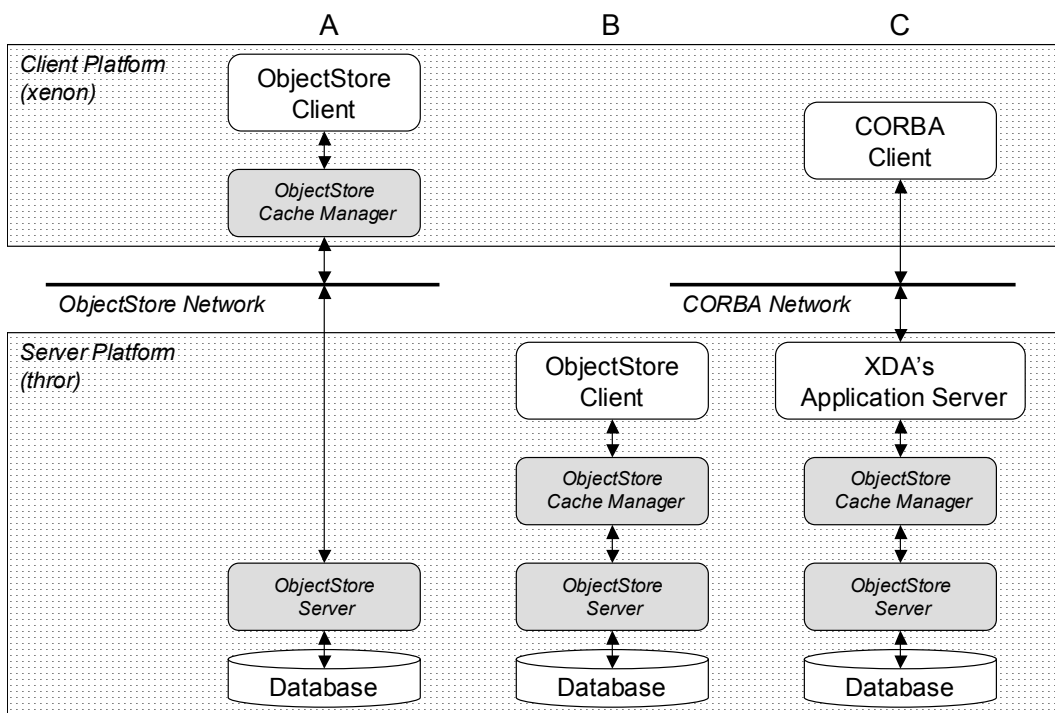


Figure 103. Test environment

All tests were performed by running a client and a server on different machines connected by an otherwise unused 100 Mb Ethernet.

The clients were running on a Sun SPARC Ultra 2 workstation (xenon). The Sun workstation had 640 megabytes of memory and was configured with two SPARC processors running with 400 MHz. All client programs running at the Sun platform were coded in C++ and compiled with the Sun’s C++ compiler version 5.2.

A Linux workstation (thror) was used for the corresponding server processes. The Linux workstation had 1 gigabyte of memory and two Pentium III processors running with 1 GHz. The CORBA's server program was also written in C++ but compiled with GNU C++ compiler version 2.95.3.

All programs were built without including debug information and any optimisations. *ORBacus* 4.0.2⁴³ [ITorbacus01] from IONA Tech. was used on the both platforms as the ORB implementation. *ObjectStore* version 6.0 SP8 from ODI Inc. [ODIug01] was used as an ODMG-conformant DBMS. The size of the ObjectStore's client cache manager was set to 983040 bytes.

7.3.3 Basic Experimental Results

The main goal of this test was to compare the performance of an original ObjectStore application with the performance of a hybrid CORBA/XDA/ObjectStore applications on the same class of tasks in order to show which one is more efficient for this particular class.

7.3.3.1 Simple Operations taking one Parameter

Figure 104 presents the performance of the `set_id` (second) operation for objects of different size. The operation was sequentially called on a given number of persistent objects. All operations were performed within one manual transaction. The figure shows the average time per operation, so the total time was divided by the number of persistent objects.

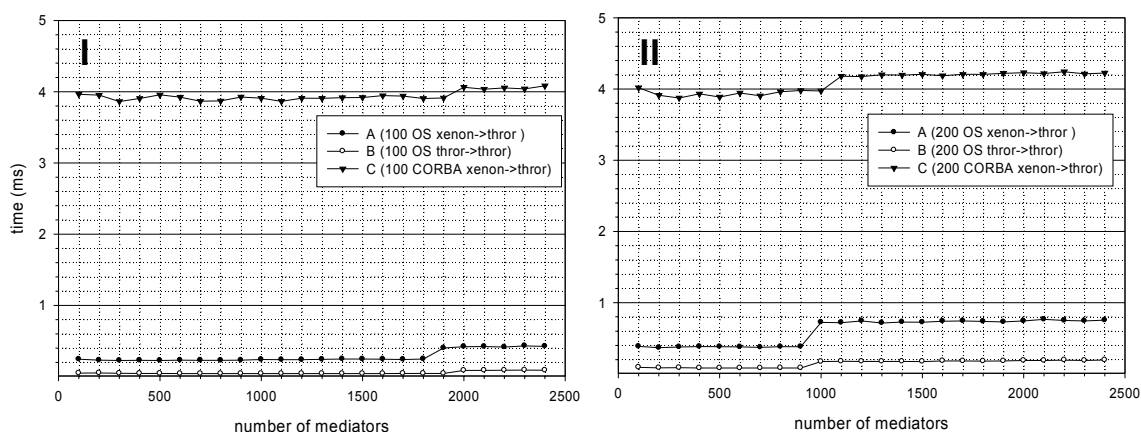


Figure 104. Performance of simple operations taking one parameter (operations `set_id` for objects of size 100*4 bytes – I, and 200*4 bytes – II)

Comparing the performance of the CORBA-based application (line C) with the local ObjectStore application (line B), we see the overhead caused by the network communication, which is about 4 msec. Comparing the CORBA-based application with the equivalent ObjectStore application (line A), we see a performance overhead ranged from a factor of 19.5x to 10x for objects of the size 100*4 bytes, and a factor of 10x to 6x for objects of the double size.

It is the worst case for the integrated CORBA/ODBMS applications. Clearly, comparing the distributed CORBA program issuing a lot of remote calls with the equivalent native ObjectStore application, we see that the first has a significant overhead. The last successfully uses a local cache manager, which caches all remote invocations. The efficiency of this cache

⁴³ Initially developed by OOC Inc., now ORBacus is distributed by IONA Tech.

shows two little „jumps“ marked at 1800 and 900 mediators, which happen, because the size of all objects accessed exceeds the size of the cache.

The test shows the ways to the design of more efficient distributed systems. It is, therefore, either to make fewer calls or to built a local cache. For small parameters up to few hundred bytes the cost of a single invocation is essentially expensive. So the application must try to make fewer calls by sending more data with each of them. To study this aspect further, we refer to Section 6.5 presenting some details about this approach.

7.3.3.2 Operations taking „bulk“ Parameters

To study this aspect, we have taken a „bulk“ version for the operation `set_data`. It reduces the messaging overhead, packing parameter values for all operations into one „bulk“ parameter `XDA_DataRepSeq`. Figure 105 compares the performance of different implementations for this operation.

The message length adds a linear delay to the latency of remote CORBA operations (comparing line C with line B). However, this delay is not as big as for the remote ObjectStore application (comparing line A with line B). Here we see a definitive 1.5x – 2.1x advantage for the CORBA-based version. The difference in the performance of the client’s and server’s hardware should not play any role for this test, since the operation does not perform any complex calculations, just only sends the data.

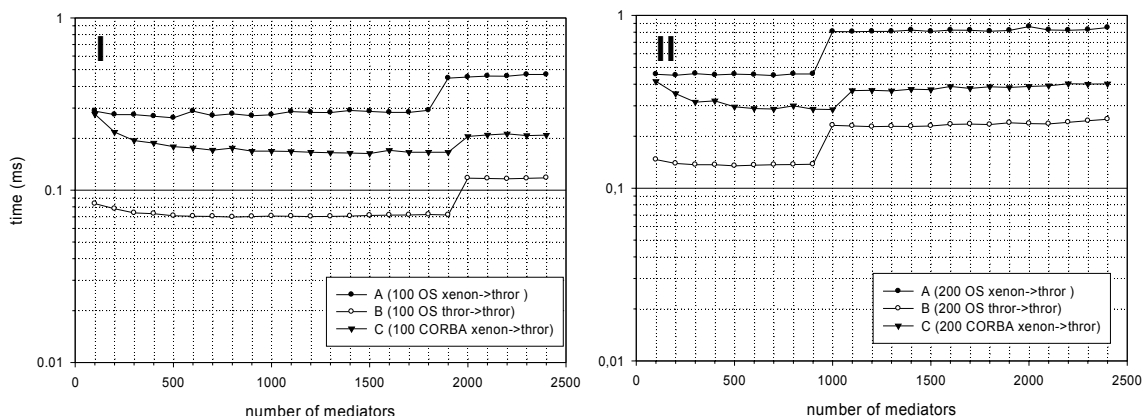


Figure 105. Performance of operations taking „bulk“ parameters (operations `set_data` for objects of size 100*4 bytes – I, and 200*4 bytes – II)

7.3.3.3 Data-intensive Operations

The actual performance of ODBMSs significantly depends on the following two factors such as clustering and client-side caching. Good clustering means that objects or pages that are frequently needed together are located next to each other on the storage system. Clustering can affect the performance on small objects significantly. Client-side caching tries to keep objects or pages in the local main memory to avoid another retrieval if they are needed again later. Thus, caching results in additional administration overhead to insure the consistency of data after updates. Client-side caching is also a major reason why ODBMS performance depends much more on the client’s hardware than RDBMS performance.

To emulate operations of the class above presented, operations performing the multiplication of arrays are used. Figure 106 shows performance of these operations for the ObjectStore and CORBA-based applications.

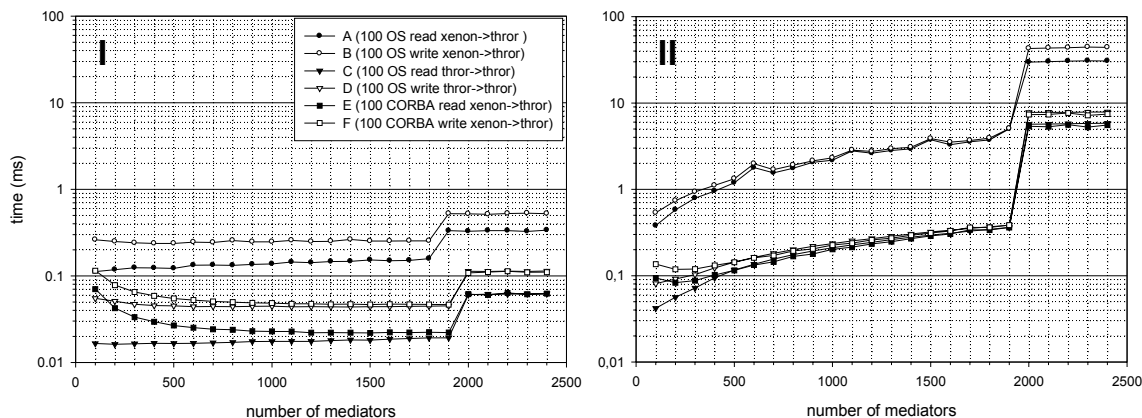


Figure 106. Performance of data-intensive operations
 (I – operations multiply_read/write_good,
 II – operations multiply_read/write_bad)

On the „good“ operations the advantage for the CORBA-based application ranged from a factor of 6x-5.2x (all data in cache) to 5.5x-4.8x (cache over), and on the „bad“ operations it is of 11x-10x (all data in cache) to 5.8x-6.1x (cache over). Here we see that the most significant advantage for the CORBA-based application is on the „bad“ operations, when the size of the ObjectStore’s cache is big enough to store all accessed objects. Moreover, the difference grows together with the increasing number of the objects.

These results can be explained by the difference in the power of the platforms and the difference in the architectures of ObjectStore and CORBA-based applications (A and C applications in Figure 103). The ObjectStore application performs all calculations on the client’s side, where CORBA-based application uses XDA’s Application Server to perform them locally on the server’s side.

For data-intensive operations involving large numbers of large persistent objects (such as our multiply operations) the messaging overhead, that is observed on the remote ObjectStore application (A), can be very large. One part of this overhead concerns the time that is necessary for the transmission of all persistent objects to the client. The other part makes up the overhead for transmitting the corresponding locking information as the operation proceeds. ObjectStore automatically (and regardless of the type of transaction) detects whether the state of a persistent object has been changed and, if yes, sends its new copy to the server, which is also a time-consuming operation.

Additionally, the performance of the both applications gets even worse, if the size of the working set exceeds the size of the ObjectStore client cache. If the current access is in the same page as the previous one, the memory location can be selected very rapidly. The data can potentially be accessed without wait states. But if a page change is required, the page must be loaded from the server before the object on the page is accessed. Changing pages in the remote ObjectStore application takes several milliseconds, and induces more wait states. In contrast to the native ObjectStore application, in the case of the CORBA-based application (C) this network overhead is much smaller. The XDA application server, which performs the role of the ObjectStore’s client, locates in the same machine as ObjectStore’s server and, therefore, no remote accesses are performed.

An interesting question is how these performance numbers compare to other systems that attempt to solve similar problems. For example, it would be interesting to compare their performance to the non-persistent system.

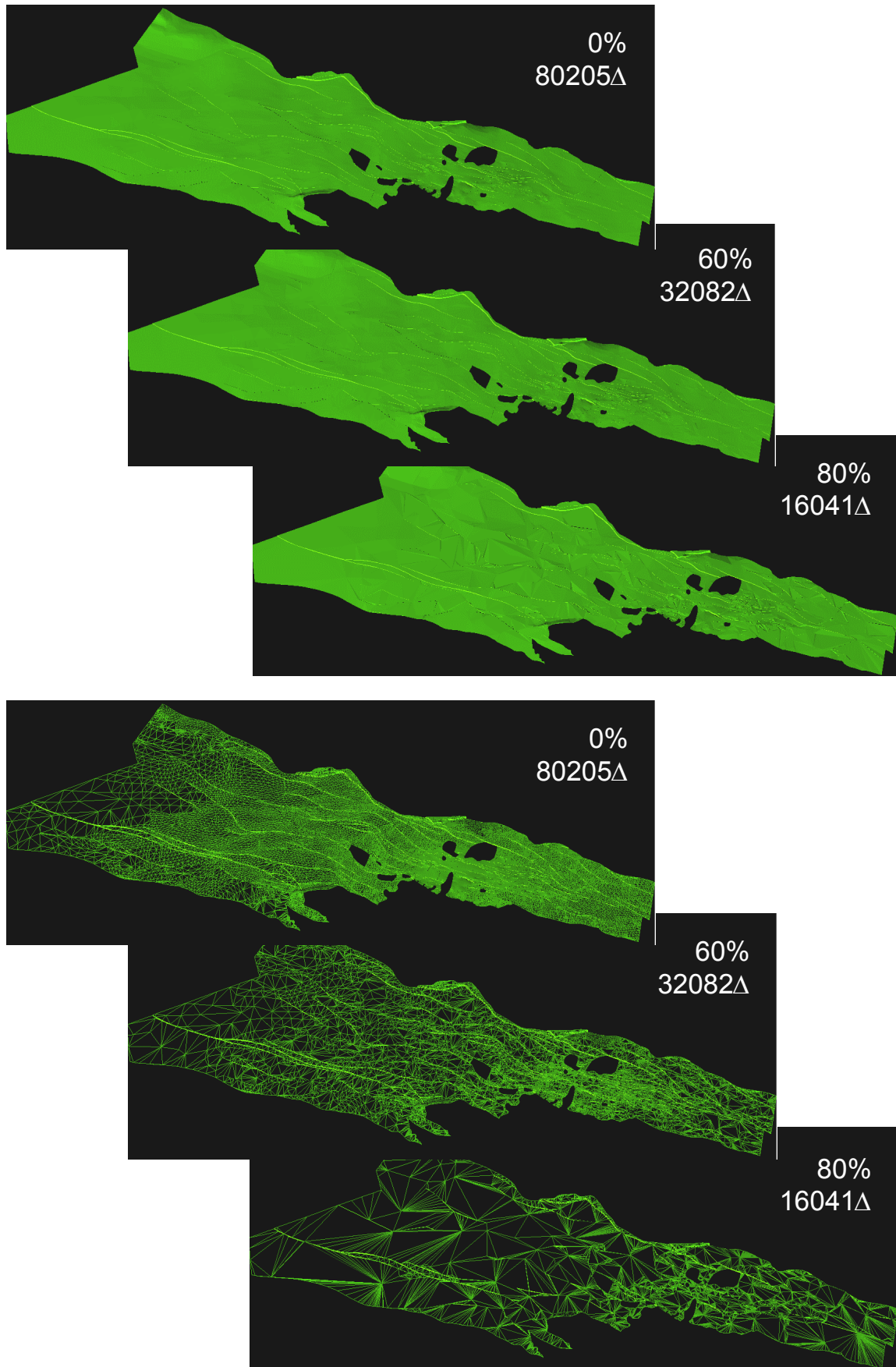


Figure 107. A stratigraphic surface with reduction factors: 0%, 60% and 80%

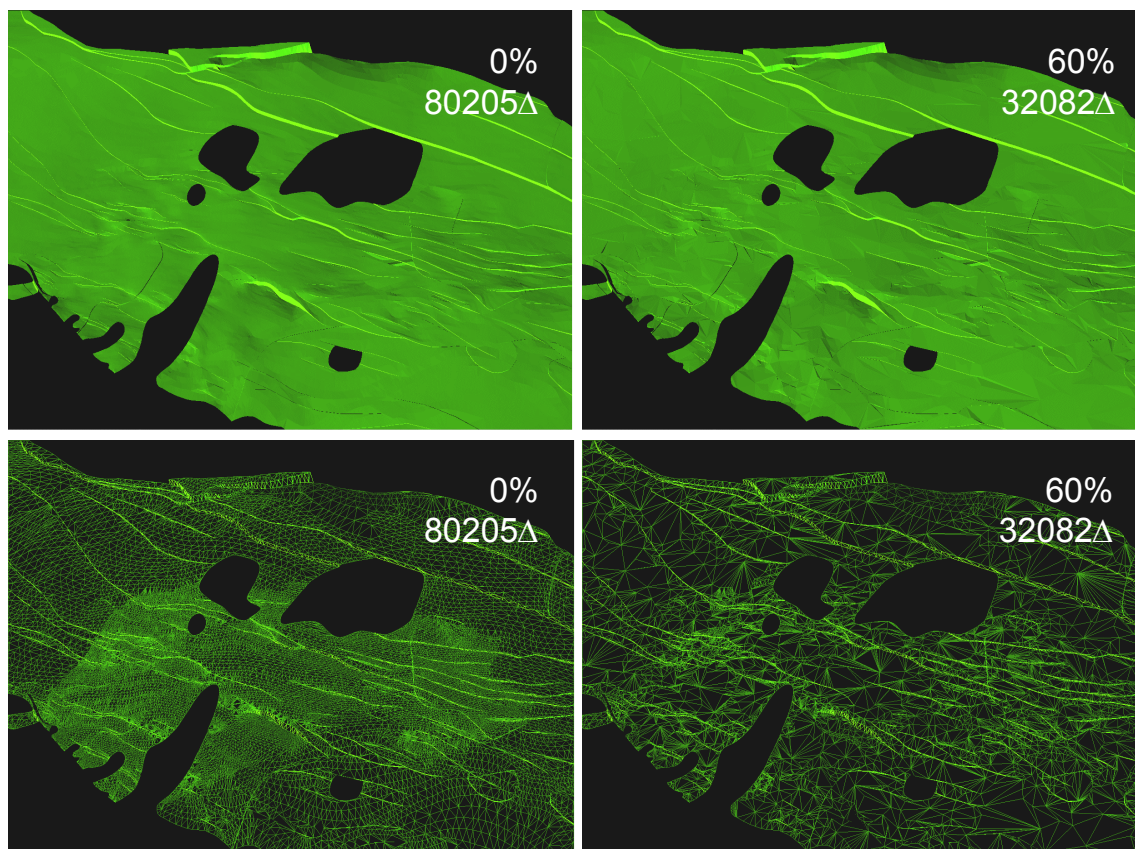


Figure 108. Part of the stratigraphic surface with reduction factors: 0% and 60%

7.3.3.4 Progressive Data Transmission

Two different datasets were used to test the usability of progressive data transmission. The first test set consisted of real 3D stratigraphic surfaces from the Lower Rhine Basin provided by RWE Rheinbraun AG. The surfaces were modelled at Rheinbraun from different types of real geological data: geological section profiles, wells, etc., and converted for subsequent editing with the GOCAD 3D modeller. The crossing faults were filled by additional triangles to obtain continuous surfaces. Every surface contains about 80000 triangles and has an average complexity of geometry that makes them a good test suite for decimation algorithms.

Generally, most of the surfaces are flat enough to survive a strong decimation with only minimal visible changes in geometry. Sometimes, however, they contain fault structures with relatively complex geometry. Figure 107 shows such a region in one of the surfaces with different levels of detail. Quite strong deformations in the geometry of faults are visible in the picture with a strong resolution reduction factor of 80%. The picture with a lower level of decimation of 60% looks similar to the original on the top with decimation of 0%. Tests have shown that 60% is the optimal level of decimation for our test suite in order to preserve enough details for most of the surfaces. Figure 108 presents a zoomed strong faulted part of the surface in the original resolution and with 60% decimation.

The second test dataset was a sequence of synthetic 3D surfaces generated with the help of GeoToolkit's 3D surface generator. It is a set of automatically created triangulated 3D surfaces with different realistic relief types ranging from flat to mountains. The size and the geometry of the surfaces are gradually varied from 1000 to 100000 triangles, making them a perfect test suite for testing the speed of the network data transfer.

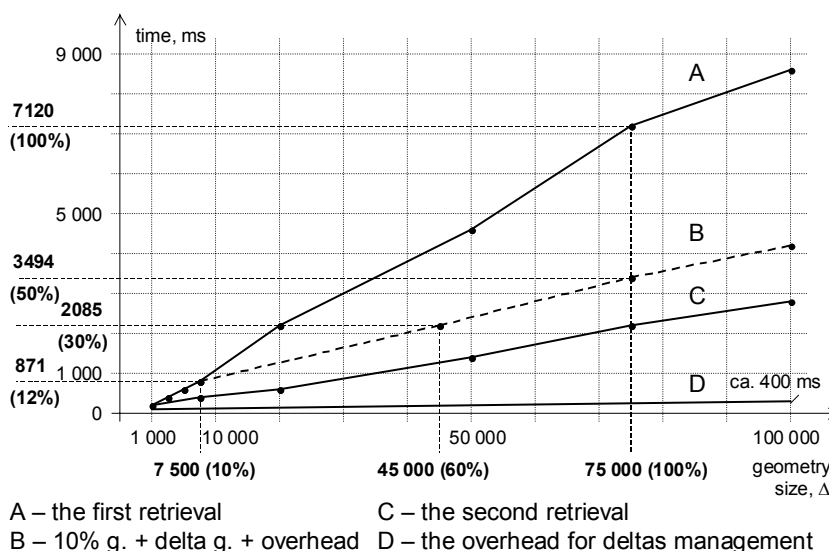


Figure 109. Speed of data transfer in relation to the object size

Figure 109 shows performance results of our study. Graph A shows the time required to read an object from the remote database. Graph C shows the time required for the second retrieval of this item. After the first time request, objects are kept in a cache, which explains the speed-up in accessing these objects a second time. Graph D shows the time required by the overhead for operations with progressively represented data on the client side. Implemented in Java, the array-based representation has an almost linear graphic with max. time of about 300ms for 100000 triangles. Finally, graph B shows the sum of the three times required to read a 10%’s geometry in the first read, a corresponding delta-geometry in the second read and the overhead in geometry conversion.

From these graphs, it is obvious that the simplification of the geometry used for pre-visualisation can significantly reduce the access time. For example, it is possible to save 88% of the time required for the loading of a surface consisting of 75000 triangles by loading its simplified 10% representation consisting of only 7500 triangles (A). Of course, when decimated to this level, the geometry is very coarse, but this is sufficient in most cases where only a rough surface identification is required. In cases where more detail is required, the 10% representation can be improved by loading a corresponding delta. This requires significant less time than the first loading because of the pre-caching. For example, a second loading of a 50% delta geometry for our surface will require about 1100ms (C). That, together with the time required for its integration in the 10% geometry (~100ms, D), will take only 1200ms. Summarising processing time for all operations including the time for loading of the first 10% geometry, we will receive 2085ms (B). This is still only 30% of the time necessary for the loading of a complete geometry.

Chapter 8

Conclusions and Further Research

This chapter summarises the thesis and discusses some possible future developments in both technologies and their potential influence on the issues of the CORBA/ODBMS integration.

8.1 Thesis Summary

The thesis puts forward a concept of a *distributed persistent object model* supporting a soft, non-intrusive integration of existing persistent object-oriented databases with CORBA. Since a schema evolution may be a very exhausting process for large data volumes, by realising a *mediator-based approach* the model permits CORBA conformity without any modifications to existing databases and applications. It is also important that the databases, which are remotely accessible through CORBA, at the same time remain available for native database applications.

On the one hand, the introduction of intermediate mediator components enhances the *flexibility* and *adaptability* of the integrated system, as well as allows the realisation of „*object views*“ on the structure of persistent objects. On the other hand, however, it brings a burden for mediator management. Our contribution here consists in identifying the important components of the programming model for *robust mediator management*, pointing out design alternatives, and collecting those components into a well-rounded execution model.

Further, eliminating the gap between the distributed (OMG) and database data models (ODMG), the model provides a single paradigm for designing distributed persistent objects. Thus, the basic advantage of the model is a property of *transparent persistence*. There is no difference between application client code working with either distributed transient or persistent objects. This is a very attractive property for „*pure*“ CORBA clients, since the model hides the overhead imposed by the persistence and makes this feature transparent. However, it also assumes the presence of efficient mechanisms for *automatic transaction control* and *server objects management*. Moreover, for advanced clients the model provides the possibility of *customising the behaviour* of these mechanisms and *access to the native database functionality*, i.e. full control of the management of databases, transactions and clustering of persistent objects. The experimental results presented in this thesis show the great influence of this model on the performance of the distributed CORBA/ODBMS system.

Based on this model, the thesis then presents the design and implementation of a customisable and efficient framework – *The eXtensible Database Adapter (XDA)* – which simplifies the development of the middle-tier level, which mediates between the input database application and CORBA. The framework not only gives the necessary *guidelines* for building *scalable well-performing* CORBA/ODBMS applications, but also reduces the cost of their development by providing *adaptable*, pre-defined *ready to use components*. Experimental results have shown that some additional techniques, such as „*bulk*“ operations, can significantly improve the performance of integrated applications.

Two alternative approaches aimed at simplifying the development of the ODBMS adapters are studied: *dynamic mediators* and *automated code generation*. The XDA *Metaobject Protocol (MOP)* provides a safe and uniform mechanism for changing software. Using dynamic mediators, the server code can be made resistant against schema changes by the following procedure. When informed of a schema change, the adapter queries the data dictionary of the database to

acquire the updated scheme. However, this flexibility is achieved at the cost of performance. The reflective operations of the MOP are slower than the equivalent non-reflective operations. Moreover, modifications at the meta level should be processed very carefully, since even the safest looking changes to the schema information in a multi-client environment may cause serious damage to the database or make it incompatible with other database applications.

The *XDA Code Generator* (CG) presents another method of simplifying the development of mediators by generating them using database schema information. Since the IDL and ODL languages are not rich enough to keep all information about database classes that is necessary for the generation of mediator classes, an extension of the ODL – *eXtended ODL* (XODL) is proposed. The XODL language introduces new constructs in specifying access type information for attributes and operations. Automatically generating mediator classes and their corresponding IDL interfaces, the CG radically simplifies the integration of existing databases with the CORBA environment. Instead of tedious and error-prone hand coding, the programmers specify their preferences, desired clients' perspectives or „*views*“, as well as implementation details concerning the mediator classes' layout as input information to the CG. Another essential component of the CG's design is the incorporation of XDA mechanisms providing additional flexibility and adaptability to the generation process and the use of CG not only for the integration of existing databases, but also for the development of new databases, which are CORBA-conformant.

Evaluating the prototype on some real applications, the thesis identifies the *crucial problems* appearing in the integration process and proposes new approaches for *efficient request dispatching* and *transaction management*. Providing transparency of persistent objects, the XDA realises efficient automatic management of database-specific functionalities, such as transactions and clustering. At the same time, the adapter provides to advanced clients the possibility of *customising the behaviour* of these mechanisms and access to the *native database functionality*, for example to control the databases, transactions and clustering of persistent objects manually. This new approach for efficient, persistence-aware request dispatching has allowed significantly improve the *performance* and *scalability* of the integrated system on databases with large numbers of persistent objects. Corresponding experimental results are presented.

Furthermore, the XDA framework is unique in that it permits the development and practical evaluation of new mechanisms aimed at improving the performance of CORBA/ODBMS systems. A long-term objective is the provision of an experimental environment for research on the resolution of semantic heterogeneity.

The XDA is a running prototype developed as a part of this thesis. At the time of writing, the XDA supports the ObjectStore DBMS. The prototype has been successfully applied in several geological research projects concerning the development of *large 3D kinematic models* of selected regions within the Lower Rhine Basin. In contrast to traditional monolithic GIS, the architecture presented has an open, distributed infrastructure that integrates the existing heterogeneous geoscientific tools into one distributed system in which the unique analytical and modelling facilities of these highly-specialised tools are coupled with efficient data storage, navigation and retrieval. This synergy of technologies makes up an essential basis for all GIS, which are required to tackle with large complex geoscientific models. Originally developed for such systems, this technology can be also applied to other applications.

We expect that the enhanced facility in handling large sets of complex persistent data provided by the techniques presented will enable us to create more comprehensive and powerful distributed CORBA/ODBMS systems.

8.2 Applicability of the XDA Approach

Open object-oriented XDA-based stores are suitable environments for any applications that need to manage distributed persistent data. Classic application areas are those involved in the management of complex data. Commonly cited examples are opening of legacy systems, development of Computer Aided Software Engineering (CASE) tools and Computer Aided Design and Manufacturing (CAD/CAM), office automation, document preparation and other distributed object-oriented systems. Presented here are some examples of possible applications of XDA.

8.2.1 Opening legacy Systems

Many large institutions are held back by existing information systems developed some years ago. These systems are mission-critical and must remain operational at all times; they are typically large and written in a legacy language, and use legacy database management systems. These systems pose a serious problem for large organisations [BS93, BS95]. A fundamental requirement for new platforms is to support legacy system migration such that the migrated target system remains fully operational in new environments.

The XDA provides a distributed object-oriented approach to integrating legacy database applications and enables the developers to write an application server that works as an objects wrapper and encapsulates the legacy system. The XDA framework architecture makes the developer's job relative simple, and as result, we have been able to produce XDA application servers for new data sources in a matter of days instead of weeks and months. The approach has been proved very efficient and effective [SB00, SS01, STCK02].

XDA's code generator represent further step to the automation of this development process. Generation of XDA's mediators is a generic task, which is not limited exclusively to CORBA integration with Object-Oriented Databases only. A wide range of legacy applications can use this approach for integration with CORBA environment. Consequently, the issues and tradeoffs of the generator's design are basically the same regardless the concrete type of legacy application being integrated. As a result the problems experienced and solved during the generator design and implementation are not ODBMS-specific and concern much wider range of legacy applications.

8.2.2 CASE and CAD Systems

Computer Aided Software Engineering (CASE) and Computer Aided Design and Manufacturing (CAD/CAM) systems are intended to automate development of large applications in conjunction with structured software engineering techniques.

An environment of typical CASE system includes the following components [SM93].

- A development workstation
- Front-end tools for planning through design
- Back-end tools for generating code
- An information repository

The information repository is claimed as the most important element of the system and all the other components will depend upon it. The information repository stores and organises all the information needed to create, modify and develop a software system. This can include data structures, processing logic, business rules, source code and project management data. Most of this data is not regularly structured and so not suitable for storage in a traditional DBMS. Furthermore, a file system does not provide the necessary management facilities.

Therefore, an object-oriented store powered by modern middleware facilities through an XDA-like adapter is an ideal environment for managing such a system. [LLOW91] cite an example of Lucid Inc. and their CASE product, Cadillac.

8.2.3 Geographic Information Systems

Geographic Information Systems (GIS) have drawn significant research interest in recent years. One of the important characteristics of these applications is the need to store and process vast amounts of irregular data. The overall rate of collection increases rapidly with advances in technologies such as high resolution satellite-borne imaging systems and global positioning systems, and with the growing number of people and organizations who are collecting and using geodata. That number will continue to grow with the growing awareness among information technologists that indexing data by location is a fundamental way to organise and use digital data. Therefore, integrating geodata from various sources is increasingly important because of growing environmental concerns, pressures on governments and businesses to perform more efficiently, and simply because of the existence of a rapidly growing body of useful geodata and geoprocessing tools [BK97].

Applicability of the XDA Approach for GIS was highly illustrated in the Chapter 7. There are also some other examples of similar systems. An Environmental Information System incorporating GIS, the World Wide Web, and CORBA is presented in [KKN+96]. A federation architecture of the system provides the functionality of Geographic Information Systems using off-the-shelf WWW browsers. CORBA is used to overcome some of the current limitations of the World Wide Web. The system supports access only to relational database systems.

An other example of a GIS built on top of an object-oriented database system is Paradise (PARAllel Data Information SystEm) [DKL+94]. It is a GIS built using the SHORE [CDF+94] object store. The system provides an extended-relational data model and uses a query shipping client-server architecture.

8.2.4 Information Systems for Medicine and Biology

Due to advances in technology, instruments used in scientific laboratories get more complex and more automated. Scientists, like physicists, chemists, and biologists who use these instruments produce more and more data in shorter periods of time. Their requirements concerning quality, correctness and accuracy of data are increasing. New techniques have made it possible to process and store data in a way and at a cost never before achievable [RD94]. Since the scientific data produced in experiments and exploited in software for their analysis is rapidly taking on a distributed nature, collaborative tools that can incorporate all types of data including sound, images, and video in an integrated manner even though the sources of data represent a first-hand interest in the modern research community.

Accessible on a wide range of platforms CORBA represent the premier open architecture, which can be used as the base for the development of such distributed computing infrastructures. One exciting example of CORBA being applied in medicine is in patient records in health care. As part of its „Science Serving Society“ mission, Los Alamos National Laboratory (LANL) has developed an open collaborative health-care environment system called OpenEMed [OM03] (formerly known as TeleMed [Fors97]). OpenEMed is a distributed medical-record system that supports image, audio, and graphical data. Utilising an entirely Java/CORBA based architecture, it integrates complete patient records with detailed radiographic data, and allows the remote sharing of patient and radiological data over networks.

The unique aspect of TeleMed is that the information really is not maintained by one ODBMS, but distributed among several databases in the network that are recognised as autonomous data sources. Thus, the system incorporates many object-oriented databases connected by the common CORBA Bus. These databases do not have to be as large as ordinary relational databases and basically hold many sets of patient objects in which many other objects, varying from static x-rays to dynamically changing heart-monitors, are connected to. All the information is distributed and integrated together throughout the TeleMed CORBA network even though to the user, all the information seems to be all in one central server.

8.2.5 Telecommunications Management Networks

Distributed object stores have also been successfully used for network management systems. British Telecom designed a system to allow manipulation of network nodes and devices [Ricc93]. C++ objects were used to model the network components. The chosen system was the Versant ODBMS. So successful was the project that a second system, also using Versant, soon followed.

A typical feature of telecommunication networks is the huge amount of managed objects. Since the ODBMS supports millions of fine-grained objects used by the manager, it must provide a fast access to the managed objects that are located both within the manager and its clients. Here application of new approaches such as XDA for efficient, persistence-aware request dispatching is crucial. It directly determines the *performance* and *scalability* of the integrated system on databases with large numbers of persistent objects.

8.3 Perspectives and Future Work

During the recent years CORBA/ODBMS integration patterns and tools have made notable progress and have become widely used by database community. But what will happen tomorrow? As both CORBA and ODBMS technologies are relatively young, they keep intensively evolving. So how will future changes in both technologies influence the way ODBMS and CORBA integrate? How will CORBA/ODBMS integration look in several years? In subsequent sections some selected future developments in both technologies will be presented as well as the degree of influence they are likely to have on CORBA/ODBMS integration process will be evaluated. The feasible evolution of CORBA/ODBMS integration tools is illustrated in Figure 110.

The most existing CORBA/ODBMS integration tools are specific to one distinct database and one distinct CORBA implementation (e.g. [IToosa97, ITova97, ITo296, Pers96]). Typically, CORBA implementers offer object database adapters, whereas ODBMS vendors produce different front-end tools. Still, in the last period a tendency towards systematisation and generalisation of CORBA/ODBMS integration tools is clearly observable. The ODAF showed the way to database adapters linking one distinct CORBA implementation (Orbix) with different databases (ODMG-compatible). In turn, our XDA prototype now integrating one distinct ODBMS (ObjectStore) with different CORBA implementations has a potential to be adjusted to work with other object-oriented and object-relational DBMS. This course is likely to be continued in the nearest future.

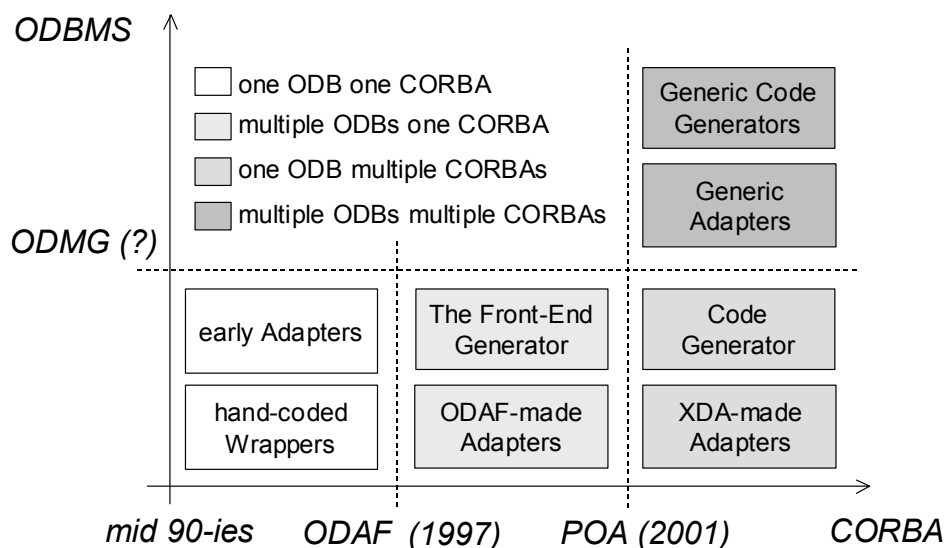


Figure 110. Past and predicted evolution of CORBA/ODBMS tools

The introduction of the *Portable Object Adapter* (POA) has brought many features important for the CORBA/ODBMS integration. Their significance is illustrated in this work under comparison of two versions of the XDA: the old BOA-based version is compared with the new POA-based version. The POA removes limitations of the old under-specified and uncompleted BOA and introduces many innovations presented in the Section 6.1. The most important of them is probably the standardisation of interfaces for the basic adapter components (e.g. responsible for request dispatching) and their interchangeability. Thus, the POA is able to „cooperate“ with server applications and allows realisation of generic database adapters such as XDA, which provide their own database-specific mechanisms for object management. Further, the usage of the standard POA interfaces also assumes the portability of these extensions between different ORB systems. Assuming the dominance of CORBA products compatible with the standard's release 2.3 (defining the POA), it is reasonable to expect the success of such generic XDA-style multi-ORB adapters in the market.

The emergence of generic XDA-style adapters makes the next step continuing the movement to common compatibility further in the direction of database systems. Since database adapters are typically products of CORBA vendors (hence each ODA is vendor-specific), the emergence of generic adapters working with several DBMSs is rather questionable. From this point of view approaches based on the code generation can be more flexible than generic adapters. Through database-specific settings a *code generator* (CG) can be adjusted to generate the code for a particular DBMS. Therefore, hybrid solutions adapter/CG have greater potential than simple adapters. From this perspective the next most expected feature of CORBA would be the specification of *reverse-mapping rules* to IDL for common programming languages⁴⁴. It would fill the remaining link in the ODBMS→CORBA migration chain: *multiple ODBMSs (ODMG meta model) → Programming Language to IDL mapping → multiple CORBA products (POA)*. In fact, some work has been conducted to standardise Java to IDL mapping [OMGj02]. If such a specification will be defined for the other programming languages (first of all, for the most popular C⁺⁺), the mapping chain will be fully standardised, which will further boost the development of generic CGs.

⁴⁴ Here, the programming language must be one of those languages supported by both ODMG and CORBA standards (i.e. C⁺⁺, Java, Smalltalk).

Nevertheless, when the ODMG standard [CB99] becomes fully supported by many ODBMSs vendors, a single generic adapter supporting several ODMG-compliant ODBMS can be realised also. For example, the core of the XDA's architecture is a database-independent virtual machine performing mediator management. It's present logic is tuned-up only for ObjectStore interface that prevents it from utilisation with arbitrary DBMSs. If all ODBMSs would support the same standardised interface, through which persistent objects can be accessed and manipulated, such a generic virtual machine for their management can be implemented on the basis of the proven design of the present XDA architecture. Almost now, since all ObjectStore's features used in the XDA are typical for many ODBMS products and have their equivalents in the ODMG standard, the architecture and techniques developed in the XDA can be applied for implementation of new generic adapters. Using the developed XDA prototype in some research projects to access real geospatial databases, we have identified aspects that are crucial for successful CORBA/ODBMS integration and proposed a set of corresponding solutions. The proposed approaches for efficient *persistence-aware request dispatching* and *transaction management* allowed significantly improve the *performance* and *scalability* of the integrated CORBA/ODBMS system. Further development these techniques will be certainly influenced by the need for improvements in efficiency of integrated CORBA/ODBMS systems on databases with large numbers of persistent objects. Therefore, these issues will remain one of the most important areas where more and more refined techniques will appear [AZ00].

Advanced techniques such as *dynamic mediators* proposed in our XDA architecture also can be taken into consideration. In the last time reflective techniques found a strong resonance in the research community [KRB91, KP94]. For example, research on the topic of dynamic platforms that would allow incremental building of distributed applications for *CORBA Component Model* [MMG00, OMGcm02]. If the ODMG standardises the *metaobject protocol* specifying a general database schema interface, then a generic adapter dynamically operating with any ODMG-compliant ODBMS can be realised. Therefore, when the ODMG standard becomes widely accepted in database community, fully generic database adapters as well as CGs will possibly emerge.

Further research direction will address remaining thorny issues in the implementation of persistent objects – it's complexity and difference between application's and database data/object-models. The XDA model is designed to support ODBMS/CORBA integration by integrating persistent capabilities of the database direct into CORBA runtime environment of the application as part of the programming language (either at runtime or if possible, at compile time). On this example we advocate the superior model of *orthogonal persistence*, and demonstrate how a loose version of this principle – *transparent persistence* can be supported for CORBA objects. Current OMG and ODMG models provide only limited support for *orthogonal persistence*, which could, however, solve some problems identified with the standard persistent services in CORBA. Such an approach would aid reasoning about the semantics of persistent applications constructed using CORBA and, therefore, make the design and programming of distributed applications easier and less error prone [LDH99, LDH00].

The supported by the XDA property of *transparent persistence* is a loose version of the *orthogonal persistence* [ABC+93], which could be reached only when *persistence by reachability* strategy is used. This strategy offers maximal flexibility since objects can change arbitrary their lifetime status at runtime, but often comes at cost of performance of the system [HC99]. The principle of persistence designation means that any allocated instance of a type is potentially persistent, so that programmers are not required to indicate persistence at object allocation time. By unifying the persistent and transient object address spaces, such that any given reference may refer to either a persistent or transient object, systems supporting orthogonal persistence have to be able to distinguish persistent objects in runtime. In systems that

support garbage collection, this persistence designation is most naturally determined by reachability from some set of known roots with the help of a *garbage collector* [Jon96]. Appearance of other, more efficient approaches supporting *persistence by reachability* strategy is expected.

Furthermore, the superior software engineering support that orthogonal persistence provides should not be withheld simply on the basis of prejudice against its reachability-based approach. The future work in this area will be concentrated around efficient approaches for including orthogonal persistence into a *standard programming language*, without changing the base language, while trying to keep the extensions to the language at a minimum. First examples of such extensions are already available. For example, *PJama* [PJama97, JA98, AJ00] and *PM3* [HC99] are orthogonally persistent extensions of Java and Modula-2 programming languages, respectively. In fact, there is also no technical reason why *more accepted systems programming languages* such as C++ cannot be retrofitted with orthogonal persistence, as opposed to the non-orthogonal realisations of persistence currently imposed on them.

Moreover, looking at the example of the PJama and similar projects it is easy to imagine the property of orthogonal persistence in a *distributed context*. The ODMG Group has already made the first step in this direction by adopting some relevant to orthogonal persistence properties in a binding of its object model to the Java programming language [CB99]. The binding allows many implementation-specific behaviours that fall some way short of the principles of orthogonal persistence, particularly with respect to persistence independence which they view as infeasible. Further studies are necessary to evaluate whether this hypothesis holds out in practice.

To summarise, in the foreseeable future further development of CORBA/ODBMS integration tools will be continued. Distributed application scenarios, targeted to the WWW, are becoming commonplace in the near future. Java applets running in Web browsers and communicating via CORBA with their data servers are the approach we investigated. The development of our prototype showed that integrating multiple languages (Java, C++), the CORBA, and an ODBMS together in one application system is a viable approach. Therefore, the experience learnt from the design and implementation of the XDA can be very valuable for future works made in the domain of the both, CORBA and ODBMS technologies. The topic of further research will be connected with building of generic reusable components and frameworks for robust distributed persistent systems. That process will probably result also in emergence of more sophisticated techniques integrating together several aspects related to the persistence, such as storage granularity, data transfer approaches, transaction management and caching.

TO BE CONTINUED

Bibliography

List of own Publications relevant to the Topic of the Thesis

- [BBB+98] Balovnev, O., Bergmann, A., Breunig, M., Cremers, A.B., Shumilov, S.: A CORBA-based approach to data and systems integration for 3D geoscientific applications. In: *Proc. of the 8th International Symposium on Spatial Data Handling*, Vancouver, Canada, 1998, pp. 396-407.
- [BBBS99] Balovnev, O., Bergmann, A., Breunig, M., Shumilov, S.: Remote Access to Active Spatial Data Repositories. In: *Proc. of the First International Workshop on Telegeoprocessing*, Lyon-Villeurbanne, France, 1999.
- [BBCS00a] Bergmann, A., Breunig, M., Cremers, A.B., Shumilov, S.: Towards an Interoperable Open GIS. In: *Proc. of the International Workshop on Emerging Technologies for Geo-Based Applications*, Ascona, Switzerland, 2000.
- [BBCS00b] Balovnev, O., Breunig, M., Cremers, A.B., Shumilov, S.: Extending GeoToolKit to Access Distributed Spatial Data and Operations. In: *Proc. of the 12th International Conference on Scientific and Statistical Database Management*, Berlin, Germany, 2000.
- [BBCS00c] Bergmann, A., Breunig, M., Cremers, A.B., Shumilov, S.: A Component Based, Extensible Software Platform Supporting Interoperability of GIS Applications. In: *Cremers, A.B., Greve, K. (eds): Proc. of the Umweltinformatik/Computer Science for Environmental protection*, Metropolis-Verlag, 2000.
- [BBCS97] Balovnev, O., Breunig, M., Cremers, A.B., Shumilov, S.: GeoToolKit: Opening the Access to Object-Oriented Geodata Stores. In: *Proc. of the International Conference on Interoperating Geographic Information Systems*, Santa Barbara, CA, USA, 1997.
- [BBCS99] Balovnev, O., Breunig, M., Cremers, A.B., Shumilov, S.: GeoToolKit: Opening the Access to Object-Oriented Geodata Stores. In: *Goodchild, M., Egenhofer, M., Fegeas, R., Kottman, C. (eds): Proc. of the Interoperating Geographic Information Systems*, Kluwer Academic Publishers, 1999, pp. 235-249.
- [BCG+98] Breunig, M., Cremers, A.B., Goetze, H.-J., Schmidt, S., Seidemann, R., Shumilov, S., Siehl, A.: First steps towards an interoperable 3D GIS – an example from Southern Lower Saxony, Germany. XXIII General Assembly of the European Geophysical Society. In: *Physics and Chemistry of the Earth*, Vol.23, Nr.3, Elsevier Science, 1998.
- [BCG+00] Breunig, M., Cremers, A.B., Goetze, H.-J., Schmidt, S., Seidemann, R., Shumilov, S., Siehl, A.: Geological Mapping based on 3D models using an Interoperable GIS. In: *Geo-Information-Systems, Journal for Spatial Information and Decision Making*, Vol. 13, 2000, p. 12-18.
- [BCG+99] Breunig, M., Cremers, A.B., Goetze, H.-J., Schmidt, S., Seidemann, R., Shumilov, S., Siehl, A.: Geologische Kartierung mit GIS auf der Basis von 3D Modellen. In: *Proc. of the Umweltinformatik 99: 13 Internationales Symposium Informatik für den Umweltschutz: Umweltinformatik zwischen Theorie und Industrieanwendung*, Magdeburg, Deutschland, 1999.

Bibliography

- [BCSS03] Breunig, M., Cremers, A.B., Shumilov, S., Siebeck, J.: Multi-Scale Aspects in the Management of Geologically Defined Geometries. In: *Neugebauer, H., Simmer, C. (eds), Proc. of the Dynamics of Multiscale Earth Systems*, Lecture Notes in Earth Sciences, Vol 97, Springer-Verlag, 2003.
- [BRSC02] Bode, T., Radetzki, U., Shumilov, S., Cremers, A.B.: COBIDS: A component-based framework for the integration of geo-applications in a distributed spatial data infrastructure. In: *Proc. of the Annual Conference of the IAMG – Creation, Management, Distribution, Access and Exploitation of Digital Spatial Data*, Berlin, Germany, 2002.
- [BS01] Breunig, M., Shumilov, S.: Preparing a new generation of environmental information systems. In: *Proc. of the 15th International Symposium Informatics for Environmental Protection*, ETH Zurich, Switzerland, 2001.
- [BS95] Briukhov, D., Shumilov, S.: Ontology Specification and Integration Facilities in a Semantic Interoperation Framework. In: *Eder, J., Kalinichenko, L. (eds): Proc. of the Second International Workshop on Advances in Databases and Information Systems*, Workshops in Computing, Springer-Verlag, 1996.
- [SB00] Shumilov, S., Breunig, M.: Integration of 3D Geoscientific Visualisation Tools. In: *Proc. of the 6th EC-GI & GIS Workshop*, Lyon, France, 2000.
- [SC00] Shumilov, S., Cremers, A.B.: eXtensible Database Adapter – a framework for CORBA/ODBMS integration. In: *Proc. of the 2nd International Workshop on Computer Science and Information Technologies*, Ufa, Russia, 2000.
- [SS01] Shumilov, S., Siebeck, J.: Database support for temporal 3D data: Extending the GeoToolKit. In: *Proc. of the 7th EC-GI & GIS Workshop*, Potsdam, Germany, 2001.
- [SSC+97] Seidemann, R., Shumilov, S., Cremers, A.B., Goetze, H.-J., Schmidt, S., Siehl, A.: Construction of geological maps using GIS – A case study from southern Lower Saxony. In: *Proc. of the European Research Conference on Space-Time Modeling of Bounded Natural Domains, Virtual Environments for the Geosciences*, Kerkrade, Holland, 1997.
- [STCK02] Shumilov, S., Thomsen, A., Cremers, A.B., Koos, B.: Management and visualisation of large, complex and time-dependent 3D objects in distributed GIS. In: *Proc. of the 10th ACM International Symposium on Advances in Geographic Information Systems*, McLean, Washington, D.C., USA, 2002.

Other References

- [ABB+98] Alms, R., Balovnev, O., Breunig, M., Cremers, A.B., Jentzsch, T., Siehl, A.: Space-time Modeling of the Lower Rhine Basin supported by an object-oriented database. In: *Physics and Chemistry of the Earth*, Vol.23, No.3, 1998.
- [ABC+93] Atkinson M.P., Bailey P.J, Chrisholm K.J, Cockshott W.P, Morrison R.: An approach to Persistent Programming. In: *Computer Journal* 26 (4), 1983.
- [ABD+89] Atkinson M.P., Banchillon F., DeWitt D., Dittrich K., Maier D., Zdonik S.: The Object-Oriented Database System Manifesto. In: *ALTAIR Technical Report*, No. 30-89, GIP ALTAIR, LeChesnay, France, 1989.

- [AJ00] Atkinson, M, Jordan, M.: A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. *Sun Labs Technical Report TR-2000-90*, Sun Microsystems Laboratories, 2000.
- [Alag97] Alagic, S.: The ODMG Object Model: Does it Make Sense?. In: *Proc. of 12th Annual Conference OOPSLA'97*, ACM, 1997, pp. 253-270.
- [Alex01] Alexandrescu, A.: *Modern C++ Design: Genetic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [AM95] Atkinson, M., Morrison, R.: Orthogonally Persistent Object Systems. In: *VLDB Journal* 4 (3), 1995, pp. 319-401.
- [Ami98] Amirbekyan, V.: *Integration of Object Databases with CORBA Environment*. PhD Thesis, University of Mining and Metallurgy in Krakow, Poland, 1998.
- [AspJ03] The AspectJ website, 2003. (<http://aspectj.org/>).
- [ASA+96] Abrams, M., Standridge, C., Abdulla, G., Fox, E., Williams, S.: Removal policies in network caches for World Wide Web documents. In: *Proc. of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, 1996, pp. 293-305.
- [AWY99] Aggrawal, C., Wolf, J., Yu, P.: Caching on the World Wide Web. In: *IEEE Transactions on Knowledge and Data Engineering*, 11(1), 1999, pp. 94-107.
- [AZ00] Amirbekyan, V., Zielinski, K.: The Role of Transaction Management in CORBA/ODB Integrated Systems' Performance. In *Proc. of the ACM Symposium on Applied Computing*, Como, Italy, 2000.
- [Bae03] Baeurle, M.: *Konzeption und Realisierung eines Codegenerators für die Integration von CORBA und ObjectStore*. Diploma Thesis, University of Bonn, 2003.
- [Baker96] Baker, S.: ORBs and Databases: Do you need both?. In: *Object Currents*, SIGS Publications, Inc., Vol.1, Issue 5, 1996.
- [Baker97] Baker, S.: CORBA Distributed Objects using Orbix. In: *ACM Press*, Addison-Wesley. 1997.
- [Bato79] Batory, D.: On Searching Transposed Files. In: *ACM Transactions on Database Systems*, Vol.4, No.4, 1979, pp 531-544.
- [BBC97] Balovnev, O., Breunig, M., Cremers, A.B.: From GeoStore to GeoToolKit: The second step. In: *Lecture Notes in Computer Science*, Vol.1262, Springer-Verlag, 1997, pp. 223-237.
- [BCF+99] Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and Zipf-like distributions: Evidence and implications. In: *Proc. of the Infocom'99*, 1999.
- [Bee90] Beerl C.: Formal Models for Object Oriented Databases. In: *Proc of the First International Conference on Deductive and Object-Oriented Databases*, 1990.
- [Bel66] Belady, L.: A study of replacement algorithms for a virtual-storage computer. In: *IBM Systems Journal*, 5(2), 1966, pp. 78-101.
- [BK96] Buehler, K., McKee, L.: *The OpenGIS Guide: Introduction to Interoperable Geoprocessing and the OpenGIS Specification*. 3rd edition, Open GIS Consortium Technical Committee, 1998. (<http://www.opengis.org/techno/guide/>)

Bibliography

- [BK97] Busse, S., Kutsche, R.: *Objektbasierte Integration einer externen heterogenen Informationsbasis*. TU Berlin, Informatik Fachbericht 97-7, 1997.
- [BPZ99] Bajaj, C., Pascucci, V., Zhuang, G.: Progressive compression and transmission of arbitrary triangular meshes. In: *Proc. of the Visualisation '99 Conference*, San Francisco, IEEE Press, 1999, pp. 307-316.
- [BS93] Brodie, M., Stonebraker, M.: *DARWIN: On the Incremental Migration of Legacy Information Systems*. Technical Report TR-0222-10-92-165. GTE Laboratories, Inc., Waltham, MA 02254, 1993.
- [BS95] Brodie, M., Stonebraker, M.: *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann Publishers, San Francisco, 1995.
- [Bub199] Buble, A.: *Comparing CORBA Implementations*. Master Thesis, Charles University, Prague, 1999.
- [Cat91] Cattell, R.: An engineering database benchmark. In: *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray (eds), Morgan-Kaufmann, San Mateo, California, 1991.
- [Cat94] Cattell R.G.G. (editor): *Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Fransisco, California, 1994.
- [CB97] Cattell R.G.G., Barry D. (eds): *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Fransisco, California, 1997.
- [CB99] Cattell R.G.G., Barry D. (eds): *Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, San Fransisco, California, 1999.
- [CD73] Coffman, E., Denning, P.: *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
- [CDF+94] Carey, M., DeWitt, D., Frank, D., Graefe, G., Muralikrishna, M., Richardson, J., Shekita, E.: Shoring up Persistent Objects. In: *Proc. of the ACM-SIGMOD Conference*, Mineapolis Minnesota, 1994.
- [CDK+94] Carey, M., DeWitt, D., Kant, C., Naughton, J.: A status report on the OO7 ODBMS benchmarking effort. In: *Proc. of the OOPSLA Workshop on Object Database Behaviour, Benchmarks and Performance*, 1994.
- [CDN93] Carey, M., DeWitt, D., Naughton, J.; The OO7 Benchmark. *SIGMOD Record*, Vol. 22, Nr. 2, 1993, pp. 12-21.
- [CHK96] Cho, E.S., Han, S.Y., Kim, H.J.: A Semantics of the Separation of Interface and Implementation in C⁺⁺. In: *Proc. of the 20th IEEE Annual International Computer Software and Applications Conference*, Seoul, Korea, 1996.
- [CI97] Cao, P., Irani, S.: Cost-aware WWW proxy caching algorithms. In: *Proc. of the USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 193-206.
- [CFZ94] Carey, M., Franklin, M., Zaharioudakis, M.: Fine-Grained Sharing in Page Server OODBMS. In: *Proc. of the ACM SIGMOD Conference*, Minneapolis, 1994.
- [CK00a] Cho, E.S., Kim, H.J.: LOD*: An ODMG Based C⁺⁺ Database Programming Language with Class-Separation Support. In: *Information and Software Technology*, Vol.42, No.5, 2000.
- [CK00b] Cho, E.S., Kim, H.J.: Class-Separation Mechanism For Integrating ODBMSs and General-Purpose OOPLs. In: *Object Oriented Systems*, 2000.

- [CM84] Copeland G., Maier D.: Making Smalltalk a database system. In: *ACM SIGMOD Record*, 14 (2), 1984, pp. 316-325.
- [COM95] Microsoft Inc.: *The Component Object Model Specification*. 1995. (<http://www.microsoft.com/>).
- [Con98] Conway, A.: *The Evictor Pattern*. IONA Technologies Ltd., 1998. (<http://www.iona.com/Developers/Cookbook/evictor/evictor.html>).
- [DA99] Dille, J., Arlitt, M.: Improving proxy cache performance: Analysis of three replacement policies. In: *IEEE Internet Computing*, 3(6), 1999, pp. 44-55.
- [Dan99] Danielsen, A.: *An evaluation of two strategies on object persistence: evaluating PJama and Versant*. Master's Thesis, Department of Informatics, University of Oslo, Norway, 1999.
- [DD95] Darwen H, Date C.J.: The Third Manifesto. In: *ACM SIGMOD Record*, 24 (1), 1995, pp. 39-49.
- [DD96] Dogac, A., Dengi, C., et al.: A Multidatabase System Implementation on CORBA. In: *Proc. of the 6th Int. Workshop on Research Issues in Data Engineering: Nontraditional Database Systems*, New Orleans, Louisiana, 1996.
- [DD98] Dogac, A., Dengi, C., et al.: Building Interoperable Databases on Distributed Object Management Platforms. In: *Communications of the ACM*, 1998.
- [Dick97] Dick K., Barry D.: *Trajectory Analysis: ODBMS Vendors*. Barry & Associates Inc., 1997. (<http://www.ODBMSfacts.com/trajectory/trajectory.html>).
- [DFMV90] DeWitt, D., Fattersack, P., Maier, D., Velez, F.: A study of three alternative workstation-server architectures for object oriented database systems. In: *Proc. of the Sixteenth Very Large Data Bases Conference*, Brisbane, Australia, 1990. pp. 107-121.
- [DKL+94] DeWitt, D., Kabra, N., Luo, J., Patel, J., Yu, J.: Client-Server Paradise. In: *Proc. of the 20th VLDB Conference*, Santiago, Chile, 1994.
- [DS72] Denning, P., Schwartz, S.: Properties of the working set model. In: *Communications of the ACM*, 15(3), 1972, pp. 191-198.
- [Fors97] Forslund, D.: The Role of CORBA in Enabling Telemedicine. (LA-UR-97-1010) *Global Forum III: Telemedicine in Vienna*, Virginia, 1997.
- [GB97] Gribble, S., Brewer, E.: System design issues for Internet middleware services: Deductions from a large client trace. In: *Proc. of the USENIX Symposium on Internet Technology and Systems*, 1997.
- [GHJ+94] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [GUW02] Garcia-Molina, H., Ullman, J.D., Wildom, J.: *Database Systems: The Complete Book*. Prentice Hall, 2002.
- [HC99] Hosking, A., Chen, J.: PM3: An Orthogonal Persistent Systems Programming Language – Design, Implementation, Performance. In: *Proc. of the VLDB*, 1999, pp. 587-598.
- [Heu92] Heuer, A.: *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme*. Addison-Wesley, 1992.

Bibliography

- [Hop96] Hoppe, H.: Progressive meshes. In: *Proc. of the SIGGRAPH*, Addison Wesley, 1996, pp. 99-108.
- [Hosk95] Hosking, A.: Benchmarking persistent programming languages: quantifying the language/database interface. In: *Proc. of the OOPSLA Workshop on Object Database Behaviour, Benchmarks and Performance*, Austin, Texas, USA, 1995.
- [Hoss00] Hosseini-Khayat, S.: On optimal replacement of nonuniform cache objects. In: *IEEE Transactions on Computers*, 49(8), 2000, pp. 769-778.
- [Hoss97] Hosseini-Khayat, S.: *Investigation of generalized caching*. PhD Thesis, Washington University, Saint Louis, Missouri, 1997.
- [HPH97] Hohenstein, U., Pleßer, V., Heller, R.: Evaluating the Performance of Object-Oriented Database Systems by Means of a Concrete Application. In: *Proc. of the 8th International Workshop on Database and Expert Systems Applications*, Toulouse, France, 1997.
- [Hugh97] Hughes, E.: *Performance evaluation and improvement of object database applications*. PhD Thesis, Computer Science Department, University of Illinois at Urbana-Champaign, 1997.
- [HV99] Henning, M., Vinoski, S.: *Advanced CORBA Programming with C++*. 3rd printing, Addison-Wesley, 1999.
- [Iam99] SFB 256. *GRAPE Graphics Programming Environment*. Version 5.3, Sonderforschungsbereich 256, Institut für Angewandte Mathematik, University of Bonn, 1999. (<http://www.iam.uni-bonn.de/grape/>).
- [ITclean97] IONA Technologies Ltd.: Object Clean-up in a server using RegisterIOCallback. *The IONA Knowledge Base*, Article ID: 466.773, 1997.
- [ITev99] IONA Technologies Ltd.: The Evictor. *The IONA Knowledge Base*, Article ID: 2169.800, 1999.
- [ITloader97] IONA Technologies Ltd.: Tips for writing a Loader for objects in an Object-Oriented Database. *The IONA Knowledge Base*, Article ID: 354.409, 1997.
- [ITo296] O2 Technology: *Accessing O2 through ORBIX*. O2 Technology, 1996.
- [ITodaf97] IONA Technologies Ltd.: *Orbix Database Adapter Framework*. Release 1.0, Dublin, Ireland, 1997.
- [IToosa97] IONA Technologies Ltd.: *Orbix+ObjectStore Adapter: Programming Guide*. Dublin, Ireland, 1997.
- [ITorbacus01] IONA Technologies: *ORBacus for C++ and Java*. Release 4.1.0, Iona Technologies Ltd., Dublin, Ireland, 2001.
- [ITorbix97] IONA Technologies: *Orbix Programmer's Guide*. Release 2.3, Iona Technologies Ltd., Dublin, Ireland, 1997.
- [ITova97] IONA Technologies Ltd.: *Orbix+Versant Adapter*. Dublin, Ireland, April 1997. Dublin, Ireland, 1997.
- [JA98] Jordan M., Atkinson M.P.: Orthogonal Persistence for Java – A Mid-term Report, In proc. of the *Third International Workshop on Persistence and Java* (P JW3), 1998, pp. 335-352.

- [Jon96] Jones, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. With a chapter by R. Lins, Wiley, 1996.
- [JS02] Jentzsch, T., Siehl, A.: Kinematic subsidence modeling of the Lower Rhine Basin. In: *Geol. Mijnbouw*, 2002.
- [JZ02] Jiang, S., Zhuang, X.: LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: *Proc. of the ACM SIGMETRICS*, 2002.
- [Khos93] Khoshafian S.: *Object-Oriented Databases*. John Wiley & Sons, Inc., New York, 1993, ISBN 0471-57058-3.
- [King78] King F.: *IBM Report on the Contents of a Sample of Programs Surveyed*. IBM Research Centre, San Jose, California, USA, 1978.
- [KKN+96] Koschel, A., Kramer, R., Nikolai, R., Hagg, W., Wiesel, J., Jacobs, H.: A Federation Architecture for an Environmental Information System incorporating GIS, the World Wide Web, and CORBA. In: *Proc. of the 3rd International Conference/Workshop on Integrating GIS and Environmental Modelling*, Santa Fe, New Mexico, USA, 1996.
- [KM94] Kemper, A., Moerkotte, G.: *Object-oriented database management: applications in engineering and computer science*. Prentice-Hall, 1994, ISBN 0-13-629239-9.
- [Koo02] Koos, B.: *Verwaltung und Visualisierung großer, komplexer, zeitabhängiger 3D-Daten in einer verteilten Umgebung*. Diploma Thesis, University of Bonn, 2002.
- [KP94] Kiczales, G., Paepcke, A.: *Open Implementations and Metaobject Protocols*. Expanded tutorial notes, Stanford University, 1994.
- [KPT96a] Kleindienst, J., Plasil, F., Tůma, P.: What We Are Missing in the Persistent Object Service Specification. In: *Proc. of the OOPSLA'96 Workshop on Large Persistent and Distributed Systems*, 1996.
- [KPT96b] Kleindienst, J., Plasil, F., Tůma, P.: Lessons Learned from Implementing the CORBA Persistent Object Service. In: *Proc. of the OOPSLA'96*, ACM SIGPLAN Notices, Volume 31, No 10, 1996.
- [KRB91] Kiczales, G., Rivières, J., Bobrow, D.: *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [KRS00] Kienzle, J., Romanovsky, A., Strohmeier, A.: A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages. In: *EPFL-DI*, No 2000/335, 2000.
- [LAC+96] Liskov, B., Adya, A., Castro, M., Day, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A., Shrira, L.: Safe and Efficient Sharing of Persistent Objects in Thor. In: *Proc. of the ACM Conference on Management of Data (SIGMOD)*, Montreal, Canada, 1996.
- [LCS+99] Liskov, B., Castro, M., Shrira, L., Adya, A.: Providing Persistent Objects in Distributed Systems. In: *Proc. of the 13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon, Portugal, 1999.
- [LDAJ00] Lebastard, F., Demplous, S., Aguiléra, V., Jautzy, O.: *ObjectDRIVER Reference Manual*. Release 1.1.127, Technical report, INRIA, France, 2000. (<http://www-sop.inria.fr/cermics/dbteam/ObjectDriver/>).

Bibliography

- [LDH00] O'Lenskie, A., Dearle, A., Hulse, D.: *Orthogonally persistent support for persistent CORBA objects*. Technical Report 151, University of Stirling, Scotland, 2000.
- [LDH99] O'Lenskie, A., Dearle, A., Hulse, D.: Persistent Operating System Support for Persistent CORBA Objects. Morrison, R., Jordan, M., Atkinson, M. (eds), In: *Advances in Persistent Object Systems*, Morgan Kaufmann, 1999, ISBN 1-55860-585-1.
- [LLOW91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore database system. In: *Comm. of the ACM*, 34(10), 1991, pp. 50-63.
- [LTB98] Leser, U., Tai, S., Busse, S.: Design Issues of Database Access in a CORBA Environment, In: *Proc. of the Workshop „Integration of Heterogeneous Information Systems“*, Magdeburg, Germany, 1998.
- [LV97] Lausen S., Vossen G.: *Models and languages of Object/Oriented Databases*. Addison-Wesley, Harlow, England, 1997, ISBN: 0-201-62431-1.
- [Maff94] Maffeis, S.: A Flexible System Design to Support Object Groups and Object-Oriented Distributed Programming. In: *Guerraoui, R., Nierstrasz, O., Riveill, M. (eds), Proc. of the ECOOP'93 Workshop on Object-Based Distributed Programming*, Lecture Notes in Computer Science, Vol. 791, Springer-Verlag, 1994.
- [Mai89] Maier D.: Why isn't there an object-oriented data model?. In: *Proc. of the IFIP 11th World Computer Congress*, 1989, pp. 793-798.
- [Mall92] Mallet, J.-L.: GOCAD: A Computer Aided Design Program for Geological Applications. In: *A.K. Turner (ed), Three-Dimensional Modeling with Geoscientific Information Systems*, NATO ASI 354, Kluwer Academic Publishers, Dordrecht, 1992, pp. 123-142.
- [Mano94] Manola F.: An Evaluation of Object-Oriented DBMS Developments. 1994 Edition, *Technical Report TR-0263-08-94-165*, GTE Laboratories Inc., Waltham, MA, 1994.
- [Mart91] Martin, B.: The Separation of Interface and Implementation in C⁺⁺. In: *Proc. of the USENIX C⁺⁺ Conference*, Washington, D.C., 1991.
- [May96] Mayers, S.: *More effective C⁺⁺*. Addison-Wesley, 1996.
- [May98] Mayers, S.: *Effective C⁺⁺*. 2nd edition, Addison-Wesley, 1998.
- [MM97] Mowbray, T.J., Malveau, R.C.: *CORBA Design Patterns*. John Wiley & Sons, Inc, 1997.
- [MMG00] Marvie, R., Merle, P., Geib, J.-M.: A dynamic platform for CORBA Component-based Applications. *Poster Session at IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware'2000*, Palisades, New-York, 2000.
- [MZ95] Mowbray, T., Zahavi, R.: *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley & Sons Inc., 1995, ISBN 0-471-10611-9.
- [NNW93] O'Neil, E., O'Neil, P., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. In: *Proc. of the ACM International Conference on Management of Data*, 1993, pp. 297-306.
- [ODIug01] Object Design Inc.: *C⁺⁺ API User Guide*. ObjectStore Release 6.0 Service Pack 6, ODI Inc., 2001.

- [ODIaug01] Object Design Inc.: *Advanced C++ API User Guide*. ObjectStore Release 6.0 Service Pack 6, ODI Inc., 2001.
- [ODIcg01] Object Design Inc.: *C++ Collections Guide and Reference*. ObjectStore Release 6.0 Service Pack 6, ODI Inc., 2001.
- [ODIba01] Object Design Inc.: *Building ObjectStore C++ Applications*. ObjectStore Release 6.0 Service Pack 6, ODI Inc., 2001.
- [OHE96] Orfali, R., Harkey, D. and Edwards, J.: *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996, ISBN 0-471-12993-3.
- [OM03] Open collaborative health-care environment system called OpenEMed (formerly known as TeleMed), Los Alamos National Laboratory (LANL). (<http://tmed.openemed.net/OpenEMed/TeleMed/>).
- [OMGcm02] Object Management Group: *CORBA Component Model, Version 3.0, (formal/2002-06-65)*. Object Management Group, Inc., Framingham, MA, 2002.
- [OMGji02] Object Management Group: *Java Language Mapping to OMG IDL, Version 1.2, (formal/2002-08-06)*. Object Management Group, Inc., Framingham, MA, 2002.
- [OMGnam01] Object Management Group: *CORBAservices: Naming Service Specification, (formal/2001-02-65)*. Object Management Group, Inc., Framingham, MA, 2001.
- [OMGoma00] Object Management Group: *Discussion of the Object Management Architecture (OMA) Guide, Version 3, (formal/00-06-41)*. Object Management Group, Inc., Framingham, MA, 2000.
- [OMGots01] Object Management Group (1997). *CORBAservices: Object Transaction Service Specification, Revision 1.2.1, (formal/01-11-03)*. Object Management Group, Inc., Framingham, MA, 2001.
- [OMGoqs00] Object Management Group. *CORBAservices: Query Service Specification, Revision 1.0, (formal/00-06-23)*. Object Management Group, Inc., Framingham, MA, 2000.
- [OMGorb01] Object Management Group: *The Common Object Request Broker Architecture (CORBA)/IIOP Specification, Revision 2.6, (formal/2001-12-01)*. Object Management Group, Inc., Framingham, MA, 2001.
- [OMGorb97] Object Management Group: *The Common Object Request Broker Architecture (CORBA)/IIOP Specification, Revision 2.0, (formal/97-09-01)*. Object Management Group, Inc., Framingham, MA, 1997.
- [OMGpoa97] Object Management Group: *ORB Portability Joint Submission (Final), (orbos/97-05-15)*. BEA, DEC, Expersoft, HP, IONA, ICL, Novell, SunSoft, TID S.A., 1997.
- [OMGpor95] Object Management Group: *ORB Portability Enhancement RFP*. Object Management Group, Inc., Framingham, MA, 1995.
- [OMGpos00] Object Management Group: *CORBAservices: The Persistent Object Service (POS) specification, (formal/2000-06-21)*. Object Management Group, Inc., Framingham, MA, 2000.

Bibliography

- [OMGpos94] Object Management Group: *CORBAservices: The Persistent Object Service (POS)*. October 1994.
- [OMGpss99] Object Management Group: *CORBAservices: The Persistent State Service (PSS) specification, Revision 2.0 , (orbos/99-07-07)*. Object Management Group, Inc., Framingham, MA, 1999.
- [OMGserv96] Object Management Group: *CORBAservices: Common Object Services Specification*. Revised Edition, March 1995, Updated November 1996.
- [OMGtrad00] Object Management Group: *CORBAservices: Trading Object Service Specification, (formal/2000-06-27)*. Object Management Group, Inc., Framingham, MA, 2000.
- [OMGtran01] Object Management Group: *CORBAservices: Transaction Service Specification, (formal/2001-11-03)*. Object Management Group, Inc., Framingham, MA, 2001.
- [OOPSLA01] OOPSLA'01: *Workshop on Objects, XML and Databases*. Florida, USA, 2001.
- [OOPSLA97] OOPSLA'97: *21st Workshop Experiences Using Object Data Management*. Atlanta, USA, 1997.
- [ORS01] Othman, O., O'Ryan, C., Schmidt, D.: The Design and Performance of an Adaptive CORBA Load Balancing Service. In: *Distributed Systems Engineering Journal*, April, 2001.
- [OV91] Ozsu, T., Valduriez, P.: *Principles of Distributed Database Systems*. Prentice-Hall Inc., 1991, ISBN 0-13-691643-0.
- [Par02] Parallel Graphics Inc.: *Cortona VRML Client*. 2002.
(<http://www.paragraph.ru/>).
- [PBJ98] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In: *Proc. of the ICCDS'98*, Annapolis, Maryland, USA, IEEE CS Press, 1998.
- [PBJ+00] Paepcke, A., Brandriff, R., Janec, G., Larson, R., Ludaescher, B., Melnik, S., Raghavan, S.: Search middleware and the simple digital library interoperability protocol. In: *DLib Magazine*, 6(3), 2000.
- [Pers96] Persistence Software: *Building High Performance Distributed Applications With Persistence And IONA*. Interactive Objects Software GmbH, 1996.
- [PJama97] PJama: *The PJama Project: About persistence*, PJama Project, University of Glasgow, Department of Computing Science, 1997.
(<http://www.dcs.gla.ac.uk/pjama/>).
- [PR00] Pajarola, R., Rossignac, J.: Compressed progressive meshes. In: *IEEE Transactions on Visualisation and Computer Graphics*, 6(1), 2000, pp. 79-93.
- [Prism00] Prism Technologies: *OpenSpirit / OpenFusion*. 2000.
(<http://www.prismtechnologies.com/>).
- [RD90] Robinson, J., Devarakonda, M.: Data cache management using frequencybased replacement. In: *Proc. of the ACM Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 134-142.

- [RD94] Rieche, B., Dittrich, K.R.: A federated DBMS-based integrated environment for molecular biology. In: *Proc. of the 7th International Working Conference on Scientific and Spatial Database Management*, Charlottesville, Virginia, 1994. (<http://www.ifi.unizh.ch/groups/dbtg/dbtg-papers.html>).
- [Rev96] Reverbel, F.: *Persistence in Distributed Object Systems: ORB/ODBMS Integration*. PhD Thesis, CS Department, University of New Mexico, 1996.
- [RG00] Ramakrishnan, R., Gehrke, J.: *Database Management Systems*. McGraw-Hill, 2000, ISBN 0-07-116898-2.
- [RH97] Rodriguez-Tome, P., Helgesen, C. et al.: A CORBA Server for the Radiation Hybrid Database, In: *Proc. of the 5th International Conference on Intelligent Systems in Molecular biology*, Halkidiki, Greece, 1997.
- [Ricc93] Ricciuti, M.: *Object Databases find their Niche*. In: *Datamation*, 1993, pp. 56-58.
- [RKC87] Rubenstein, W., Kubicar, M., Cattell, R.: Benchmarking Simple Database Operations. In: *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, 1987, pp. 387-394.
- [RM97] Reverbel, F., Maccabe, A.: *Making CORBA Objects Persistent: the Object Database Adapter Approach*. In: *Proc. of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, USA, 1997, pp. 55-65.
- [Roa00] Roantree, M.: *Constructing View Schemata Using Extended Object Definition Language*. PhD Thesis, Napier University, 2000.
- [RSB+96] Rouby, D., Souriot, Th., Brun, J.P., Cobbold, P.R.: Displacements, Strains, and Rotations within the Afar Depression (Djibouti) from Restoration in Map View. In: *Tectonics*, 15(5), 1996, pp. 952-965.
- [Rupp99] Rupp, S.: *A Critique of Release 2.0 of ODMG-93 Standard*. 1999. (<http://gameboy.gia.rwth-aachen.de/odmgbugs>).
- [RV00] Rizzo, L. and Vicisano, L.: Replacement policies for a proxy cache. In: *ACM/IEEE Transactions on Networking*, 8(2), 2000, pp. 158-170.
- [RVV+00] Rische, N., Vaschillo, A., Vasilevsky, D., Shaposhnikov, A., Chen, S.-C.: A Benchmarking Technique for DBMS's with Advanced Data Models. In: *Proc. of the ADBIS-DASFAA Symposium on Advances in Databases and Information Systems*, 2000.
- [Ser90] Servio Inc.: *Programming in OPAL*. Version 2.0, Servio Logic Development Corporation, 1990.
- [She91] Sheth, A.: Semantic Issues in Multidatabase Systems. Preface by the special issue editor, In: *SIGMOD RECORD*, Vol. 20, No. 4, 1991.
- [SKS97] Silberschatz, A., Korth, H.F., Sudarshan, S.: *Database System Concepts*. 3rd ed., McGraw-Hill Inc., 1997, ISBN 0-07-044756-X.
- [SKW99] Smaragdakis, Y., Kaplan, S., Wilson, P.: EELRU: Simple and Effective Adaptive Page Replacement. In: *Proc. of the ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, 1999, pp. 122-133.
- [SLY+99] Sheu, R.-K., Liang, K.-C., Yuan, S.-M., Lo, W.-T.: A New Architecture for Integration of CORBA and ODBMS. In: *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 5, 1999.

Bibliography

- [SM93] Sprague, R., McNurlin, B.: *Information Systems Management in Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
- [Smit82] Smith, A: Cache memories. In: *Computing Surveys*, 14(3), 1982, pp. 473-530.
- [SML97] Schroeder, W., Martin, K., Lorensen, W.: *The Visualisation Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 1997.
- [SR92] Sarkar M., Reiss S.P.: A Data Model for Object-Oriented Databases. *Technical report CS-92-56*, Brown University, Department of Computer Science, 1992.
- [SRL+90] Stonebraker M, Rowe L A, Lindsay B, Gray J, Carey M, Brodie M, Bernstein P, Beech D.: Third-generation Database System Manifesto. In: *ACM SIGMOD Record*, 19 (3), 1990, pp. 31-44.
- [SSV99] Shim, J., Scheuermann, P., Vingralek, R.: Proxy cache algorithms: Design, implementation and performance. In: *IEEE Transactions on Knowledge and Data Engineering*, 11(4), 1999, pp. 549-562.
- [Str94] Stroustrup, B.: *Design and Evolution of C++*. Addison Wesley, 1994.
- [SUNj02] SUN Microsystems Inc.: *Java 2 Platform – Standard Edition*. 2002. (<http://java.sun.com/j2se/>).
- [SUNjb02] SUN Microsystems Inc.: *JavaBeans Component Architecture Documentation*. 2002. (<http://www.javasoft.com/beans/docs/index.html>).
- [Sutt00] Sutter, H.: *Exceptional C++: Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000.
- [SV9798] Schmidt D., Vinoski S.: Object Interconnections: C++ Report. In: *SIGS Publications*, Columns 11-14, October 1997 – September 1998.
- [Tasc03] Tasci, M.: *Konzeption und Realisierung einer Komponente für XDA zur Dynamischen Verwaltung persistenter Objekte*. Diploma Thesis, University of Bonn, 2003.
- [TS02] Thomsen, A., Siehl, A.: Towards a balanced 3D kinematic model of a faulted domain – the Bergheim open pit mine, Lower Rhine Basin. In: *Geol. Mijnbouw*, 2002.
- [Tsurf02] Tsurf Inc.: GOCAD. 2002. (<http://www.t-surf.com>).
- [TuBu01] Tůma, P., Buble, A.: Open CORBA Benchmarking. *Tech. Report No. 2001/1*, Dep. of SW Engineering, Charles University, Prague, 2001; in a slightly modified version is In: *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.
- [Tuma97] Tůma, P.: *Persistence in CORBA*. PhD Thesis, Charles University, Prague, 1997.
- [VA94] Vasudevan, V., Anthony, R.: *Approaches for the Integration of CORBA with ODBMSs*. 1994. (<http://citeseer.nj.nec.com/27738.html>).
- [VRML97] The VRML Consortium Inc. *International Standard ISO/IEC 14772-1:1997: The Virtual Reality Modeling Language (VRML)*. 1997.
- [W3C98] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0*, 1998. (<http://www.w3.org/TR/1998/REC-xml-19980210>).

- [WA97] Wooster, R., Abrams, M.: Proxy caching that estimates page load delays. *Computer Networks*, 29(8-13), 1997, pp. 977-986.
- [Wald91] Waldo, J.: Controversy: The Case for Multiple Inheritance. *Computing Systems* 4.2, 1991.
- [Wied92] Wiederhold, G.: Mediators in the Architecture of Future Information Systems. In: *IEEE Computer*, March, 1992.
- [Wied97] Wiederhold, G.: Value-added Mediation in Large-Scale Information Systems. In: *Meersman, R., Mark, L (eds): Proc. of the Database Application Semantics*, Chapman and Hall, 1997, pp. 34-56.
- [WT00] Wagner, S, Tari, Z.: A Caching Protocol to Improve CORBA Performance. In: *Proc. of the Australasian Database Conference*, 2000, pp. 140-148.
- [WT94] Wells, D., Thompson, C.: Evaluation of the Object Query Service Submissions to the Object Management Group. In: *IEEE Quarterly Bulletin on Data Engineering* 17(4), 1994, pp. 36-45.
- [Youn94] Young, N. E.: The k-server dual and loose competitiveness for paging. In: *Algorithmica*, 11(6), 1994, pp. 525-541.
- [ZA97] Zielinski, K., Amirbekyan, V.: *What CORBA/ODBMS integration technique to choose: Adapter vs. Wrapper*. In: *Proc. of the Workshop №21: Experiences Using Object Data Management in the Real-World*, OOPSLA'97, Atlanta, Georgia USA, 1997.
- [ZM90] Zdonik S.B., Maier D.: *Readings in Object-Oriented Database Systems*. Morgan Kaufman Inc., 1990.