DOCTOR OF PHILOSOPHY

A framework of services to provide a persistent data access service for the CORBA environment

Ball, Craig

Award date:
1999

Awarding institution:
University of Wales, Bangor

Link to publication

A Framework of Services to provide a Persistent Data Access Service for the CORBA Environment

Craig Ball

Thesis submitted in candidature for the Degree of Doctor of Philosophy

September 1999

School of Electronic Engineering and Computer Systems
University of Wales, Bangor
United Kingdom

Summary

Background: CORBA is a component architecture that enables inter-operability between software components that are based on heterogeneous platforms and in heterogeneous languages. CORBA goes further to specify a set of services providing low-level functionality to CORBA components. These CORBA services are vastly lacking in a specific area of functionality, this is accessing persistent data. The Persistent Object Service was the original service to provide this functionality, but has been discredited due to design faults and ambiguities resulting in the specification being withdrawn. This thesis presents an investigation into the analysis, design and implementation of a framework of services that together provide a persistent data access service.

Objectives: The service should satisfy the missing functionality in CORBA, therefore opposing the need for developers to implement their own proprietary solutions. Proprietary solutions are expensive in developer resources and data access code is highly dependent on the proprietary data storage mechanism being used. The service will provide insulation from proprietary data storage mechanisms by encapsulating its data access interface, thus breaking the dependency of code on the specific data storage mechanism. The type of data and data storage mechanisms used in the Exploration & Production industry and their particular characteristics will be considered in the design of the service.

Approach: The investigation has examined current methods of accessing persistent data from CORBA, including an in-depth analysis of the Persistent Object Service and a case study of a commercial application. The problems with these methods has been studied resulting in a list of requirements for the service to meet. A high-level design for the service has been outlined and a number of services have designed from bottom-up to reflect it.

Results: The principal outcome of the research is the design of the Persistent Data Access Service that allows the manipulation of complex structured entity data. This data can be resident in any datastore, but has to be mapped to a standard data definition model. Other functionality that services that have been designed and implemented are providing distributed streams permitting copy-by-value and data transfer performance enhancement, distributed access to files and the transfer of complex structured entities in a stream.

Implications: The Persistent Data Access Service gives to CORBA developers a standard means to access persistent data resident in heterogeneous datastores. This prevents applications becoming dependent on a specific data storage mechanisms and takes the responsibility away from the developer to implement their own proprietary solutions. The service also lets new CORBA applications and legacy applications run in parallel, as no changes has to be made to actual datastores and data models, unlike CORBA's future Persistent State Service.

List of Publications

C.H.Ball & S.Hope. *Data Access and Transportation Services for the CORBA Environment.* TOOLS '98 Conference, Santa Barbara, USA, August 1998.

Acknowledgements

I would firstly like to express my deep gratitude to Miss Patrina Ellison, whose love and encouragement has kept me going through the years, as well as keeping me sane. Next, I would like to express my thanks to my supervisor, Mrs. Sian Hope for setting up the Ph.D. and her valuable support during the work. I would also like to thank Steve Trythall for his part in setting up of the Ph.D. and for the great opportunity he gave me to work with PrismTech in Houston, Texas for 4 months. Wow, it was hot! Thanks to Dr. Thomas Varsamidis for being my second supervisor and for being a good friend in the last few years, I shall miss our daily discussions.

Other people I would like thank who have helped me along the way are:- Dr. Dylan Banarse, Dr. Andy Duller, Rallis Farfarakis, Ian France and Dave Whitehead.

I would also like to acknowledge SEECS and PrismTech for their financial funding to make this work possible.

For Patrina.

Contents

Chapter 1

Introduction

Software systems are increasingly being developed that consist of components, where a component is a self-contained piece of code providing a known functionality. Components can be plugged together to provide application level functionality and can also be distributed across machines and processes. A component architecture allowing this type of application building is the Common Object Request Broker Architecture(CORBA). This project focuses on CORBA, but the problems, concepts and solutions discussed are also valid for most other component architectures e.g. COM[Orfali 96], Enterprise Java Beans[SUN 98].

Traditional software systems that need to persistently store data are centred around storing their data in data storage mechanisms e.g. files and databases. This data is typically accessed using the data storage mechanism's proprietary interface and is invaluable to the corporations that own it.

In the move towards distributed component technology, these datastores must be brought to reach of the newly created component applications. This will allow the reuse of their stored data and to permit the running of already existing applications and new component applications in parallel. The aim of this research is to design a way to integrate these differing paradigms to achieve these requirements.

The following gives an overview of the CORBA component architecture, models of data access, what this integration should achieve and why it should be realised in a set of CORBA services. Also, described is the research's interest of integrating Exploration and Production industry standard datastores into the CORBA environment.

1.1 The CORBA Component Architecture for Inter-operability, Reuse and Portability

Object-Orientation(OO) is a paradigm that models real-world entities and their relationships in a logically definable manner. An object-oriented model of a problem domain can be simulated in software and solutions to the problem can be designed. One of major advantages of OO is its flexibility. Should the problem domain change, the object-oriented model can be extended or altered to reflect the change.

In recent years, Object-Oriented Technology(OOT)[Winblad 90] has come to fruition, with its wide spread use in systems analysis, design and implementation. There are numerous Object-Oriented analysis and design methodologies [Rumb 91][Coad 90][Coad 91] that act as a guide to examining a problem and designing a system for its solution. For the implementation of systems, OOT has become popular as a basis for many programming languages. Object-Oriented programming languages have enabled developers to tackle complex tasks such as building graphical user interfaces with ease. OOT is also making an impact in the data storage domain with the advent and evolution of object-oriented databases[Joseph 90]. Object-oriented databases are solving problems where tradition relational databases were very inefficient solutions such as CAD applications.

However, inter-operability between Object-Oriented systems is exceedingly poor. Each system is effectively closed. There is no standard OO communication protocol. To achieve communication between OO based systems, developers have to employ low-level communication protocols such as sockets[Stevens 90]. These low-level communication protocols do not fit well with the Object-Oriented philosophy and degrade the flexibility of OO based systems.

This was the situation until the Object Management Group(OMG)[OMG 97] was formed in 1989 by a few major software related corporations. Today, corporate membership of the OMG has dramatically increased to become the largest software standards organisation in the world.

The OMG's primary goal is to provide specifications for the inter-operability of object-based systems, where parts of the systems are distributed in terms of machines/processes and implemented on heterogeneous platforms in heterogeneous programming languages. The OMG's solution to this interoperability requirement is the Common Object Request Broker Architecture(CORBA)[OMG 95]. CORBA is a specification for inter-operability in a heterogeneous environment and raises inter-operability to the application level.

The model of the CORBA inter-operability architecture consists of objects requesting operations of the interfaces of server objects through a middleware software layer called the Object Request Broker(ORB). The key element of CORBA inter-operability is the utilisation of the OMG's Interface Definition Language(IDL) to specify the operations that a component offers to clients through its interface. IDL is object-oriented based, thus uses concepts such as inheritance of interfaces and an interface is the equivalent of an object, although the interface's implementation might not be an object-oriented object. An active piece of code or object instance implementing an IDL interface and carrying out the operations of an interface is called a CORBA object.

Currently, there is a trend towards creating software systems out of components, where a component is a unit of code providing a known functionality and also has a public interface to access its functionality. Components can then be plugged together to create whole applications. This is contrary to applications of the past that have been created using enormous monolithic bodies of code. These monolithic applications are brittle in the sense that they are difficult to change, extend and debug. Also a vast amount of their functionality is never utilised by users. Component applications solve this problem by gluing together tried and tested components with well defined interfaces and behaviour. Further, applications can be dynamically configured by including only the components serving the functionality required by a user.
A CORBA object makes an ideal component with its programming language independent interface and the transparency of location, language and platform features provided by an ORB.

The ORB acts as the glue bringing components together allowing them to communicate to carry out their tasks in a coherent manner. Thus, CORBA provides an excellent component architecture as well as an almost universal inter-operability mechanism.

One of the primary benefits of component architectures is their ability to easily reuse components i.e. instead of re-writing pieces of code, previously proved code serving the required functionality can be reused. The OMG has taken reuse to a higher level in its Object Management Architecture(OMA)[OMAG 90]. The OMA is an idealised architecture concerning how a software system should be structured. The OMA separates component applications into layers made up of low level common services, higher level application services and applications. The OMG is in the process of developing a standardised sets of services to fill the OMA layers. The specification of these services include the definition of their IDL interfaces and behaviour of the components implementing these interfaces. The specification of these standard services not only encourages reuse, but also has the benefit of enhancing the portability of applications across platforms, ORB products and service implementations.

The CORBA services are the OMG's standard set of services for the low level common services of the OMA. The implementation of these services provide much of the basic functionality needed by CORBA objects/components to operate in the distributed environment. The services cover functionality such as object creation/deletion, naming, streaming, events, concurrency and transactions. Concurrency and transactions are especially important for any reliable system.

1.2 CORBA Persistent Data Access

This thesis focuses on a vital area of functionality that the CORBA services do not adequately cover. This area is access to persistent data i.e. access to data held in some storage mechanism that is resilient to system failures e.g. databases and files.

The Persistent Object Service(POS)[OMG COSS] was the OMG's original solution to providing persistent data access from CORBA. However, POS has since been discredited as well as abandoned by the OMG due to faults and ambiguities in its design. One of the few commendable aspects of POS was its goal to support access to datastores of all types. Datastore types such as traditional datastores e.g. files and relational databases, as well as support for newer object-oriented databases. Object-oriented databases that have an object model similar to that of the CORBA object model making their integration easier. Future OMG persistent object services will be biased towards integrating datastores with models similar to that of the CORBA object model. Access to existing traditional datastores and the valuable data held within them is an essential requirement of corporations in their move towards distributed component based systems.

The framework of services presented in this thesis are designed to supply a solution to providing access to data stored in heterogeneous datastores. The services will attempt to meet this goal that POS failed at, by analysing the failures of POS and rebuilding from bottom up a set of services to meet this goal.

1.3 Models of Application Persistent Data Access

There are two general models that applications use to access persistent data. The first model is based on the traditional client-server model[Orfali 94] of an application directly accessing a datastore using the datastore's proprietary data access interface(Figure 1.1a). The principal problems inherent with this model includes carrying out all processing of data on the client machine, thus the business logic involved in data processing is deeply embedded in the application. In respect to actual data access the application becomes highly dependent on the proprietary datastore access interface. This dependency makes the application very inflexible to change in its data storage mechanism.

The second model of persistent data access is the three tier model[Orfali 96][Shan 98](Figure 1.1b) and is typical of how component systems are structured. The three tiers of the model are:-

- A top level client application tier.
- A middle tier composing of objects that contain the business logic to enable the processing of application requests. This separates the processing and the logic needed for the processing from the internal code of the client application as in the client-server model.

- The bottom tier is the data storage tier. The middle business object tier makes use of this tier to persistently store its data. This data storage tier is typically occupied by file systems, databases and industry specific standard datastores.

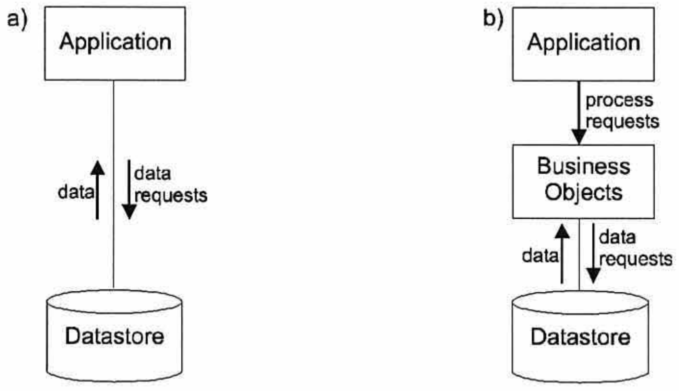


Figure 1.1: *The client-sever model(a) and the three tier model(b)*

The benefit of the three tier model is that the business logic is taken out of the application and put into the autonomous middle tier. Using this model, applications indirectly access persistent data in the data storage tier through the middle tier. This indirect data access prevents the high dependency which the application has with the datastore in the client-server model, therefore the application is insulated from the proprietary nature of the datastore and any changes made to the data storage tier. However, the high dependency is only shifted away from the application to between the middle business object tier and the data storage tier.

Software systems built using either of these models both suffer from the high dependency on a data access interface. Another goal of a persistent data access service is to decrease this dependency, so that applications/business objects are insulated from the proprietary data access interface. Thus, the service will introduce an additional tier into the models that hides the proprietary nature of datastores used for data storage(Figure 1.2).

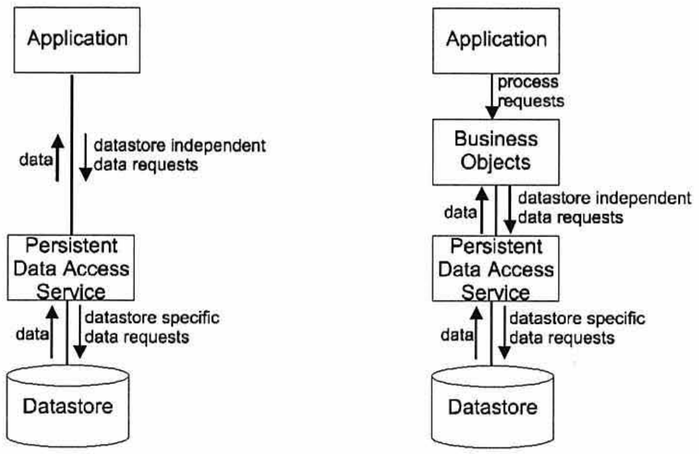

Application
data
datastore independent data requests
Persistent Data Access Service
data
datastore specific data requests
Datastore
Application
process requests
Business Objects
data
datastore independent data requests
Persistent Data Access Service
data
datastore specific data requests
Datastore

Figure 1.2: *An additional tier in the client-server model(left) and three tier model(right) to hide the proprietary nature of the datastores*

1.4 Persistent Data Access in the Exploration and Production Industry

This project is a joint venture with PrismTech[PrismTech]. PrismTech specialises in providing information systems to the Exploration and Production(E&P) sector of the Oil and Gas Industry. Information systems of the E&P industry are especially complex due to the nature of the E&P data, this varies from highly complex networks of objects to enormous arrays of scientific data that can have the additional complication of being related to spatial locations.

Initially, the project was examining the persistence mechanism of a PrismTech product called OpenBase. OpenBase was an innovative ORB product with a built-in persistence mechanism for C++ objects that are the implementations of CORBA objects. However, OpenBase did not achieve production release.

The focus of the project shifted to a new direction which was examining how instance data of the POSC Epicentre model[POSC 95] could be accessed from CORBA. POSC is an organisation producing standards to help the E&P industry integrate its information systems. One such standard is the Epicentre model. Epicentre is a data model that describes a large number of objects that represent the majority of data items that need to be stored in the E&P industry. Epicentre gives the industry a universal data model allowing all E&P organisations to share data that complies with the model.

POSC also defines an API to access and manipulate Epicentre instance data called the Data Access and Exchange(DAE) interface[POSC 95b]. The DAE is independent of any database technology, but the implementation of a DAE datastore is usually provided by a layer of software that performs a mapping between the DAE and an actual database.

To access Epicentre data from CORBA, the DAE could simply be re-written in IDL and a bridge be provided to the actual DAE datastore interface(Figure 1.3). However, this solution causes two problems. Firstly the CORBA application is still greatly dependent on the DAE interface. Secondly, the solution is inefficient as each DAE operation invocation requires a slow ORB request. This could be solved by caching data locally in the client, but this creates further problems, such as how to transport data across the ORB. Any data transported across the ORB has to have an IDL definition, and as Epicentre has around 1,500 objects, providing a compiled IDL definition of these objects is unfeasible.

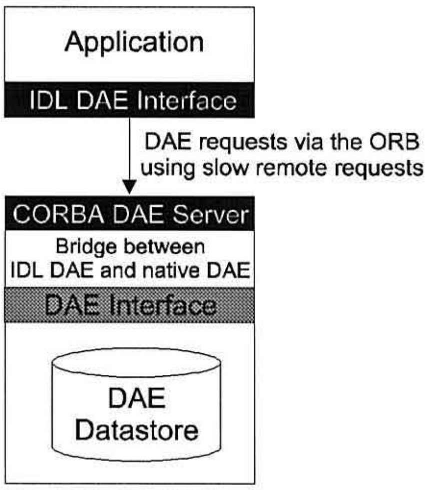

Application
IDL DAE Interface
DAE requests via the ORB using slow remote requests
CORBA DAE Server
Bridge between IDL DAE and native DAE
DAE Interface
DAE Datastore

Figure 1.3: *A DAE bridge to allow access to a DAE datastore from CORBA applications*

A CORBA persistent data access service should provide a more generic interface to accessing data than the DAE, again insulating the application from a proprietary datastore mechanism. It should also define services to enable the efficient caching of data in clients, so that it can be manipulated at local invocation speeds. This caching/transport mechanism should efficiently handle the typical types of E&P Epicentre data i.e. complex networks of objects and large arrays of scientific data. As the Epicentre model is so large it should further provide a way of handling this data without statically compiled IDL code.

Applications accessing Epicentre data via the DAE reflects the client-server model. PrismTech is also leading a coalition of E&P companies in developing a three tier architecture called OpenSpirit[Godfrey 97] to support information integration in the E&P industry. The OpenSpirit project is defining and implementing middle tier business objects representing common E&P entities e.g. wells and wellbores. The attribute data of these business objects are persistently stored in various popular E&P datastores including Epicentre and the DAE.

In fact, the mapping between the OpenSpirit objects and Epicentre was personally carried by the author of this thesis. This gave a good insight into the complexities of manually mapping between differing data models such as:- the large amount of code needed to perform a relatively simple mapping, the high possibility of bugs in this code, representing object references between models and the code's high dependency on the DAE interface.

The OpenSpirit objects are fairly coarsely grained. On the other hand Epicentre is very finely grained, hence many Epicentre objects would make up the data of a single OpenSpirit object. For the relatively few OpenSpirit objects currently being developed manually, mapping them to Epicentre is just about a feasible task, but should the complexity of OpenSpirit model increase then the manual mapping would quickly break down as a result of the exponential increase in complexity of the mapping code.

Ideally, mapping between two models should be an automated process with the only human interaction being to specify the rules to map between the models. The primary problem with automated mapping is accessing the data in the source and target models, providing a common data access interface to source and target models greatly simplifies automated mapping. Hence, another role which a persistent data access service could fill.

1.5 Aims of the Research Work

The aim of this work is to design a set of CORBA services to provide access to heterogeneous datastores, including the ability to cache data locally for fast manipulation. The ability of the service to efficiently handle Exploration and Production(E&P) industry data should also be considered in the design of the service.

The design of these services is achieved by meeting the following objectives:-

- Analysis of the current CORBA data access solutions.
- Analysis of the Persistent Object Service and taking its faults into consideration.
- Analysis of future methods of CORBA data access.
- Familiarisation with the Epicentre model and its Data Access & Exchange(DAE) interface and its special needs.
- Specifying a set of requirements that the services should meet.
- Creating a high level model of how these services should work and interact, including its integration with the CORBA transaction and concurrency services.
- The design of the IDL interfaces to these services.
- Sample implementations of the designed services.
- Analysis of how the services meet their requirements, especially in relationship to the Epicentre and the DAE.

1.6 Overview of the Thesis

Chapter 2 presents an introduction into the background standards and technologies that are essential to have an understanding of this thesis. Covered in detail is the CORBA architecture, as well as the OMG CORBA services that are relevant to comprehending the services that are the solution to the set problem. Also included is an overview of the Epicentre model and its Data Access & Exchange interface, the EXPRESS information modelling language which Epicentre is defined in and the STEP project that created EXPRESS. The chapter concludes with a discussion on the differences between CORBA on one hand and the Epicentre/DAE & STEP architectures on the other.

Chaper 3 provides an analysis of current solutions of achieving data access from CORBA. The analysis includes a discussion on the problems incurred by directly using proprietary storage technologies to persistently store data. The OpenSpirit project is given as an example of a three tier architecture reliant on proprietary storage technologies. The analysis also covers standardised methods of persistent data access including:- an extensive examination of the Persistent Object Service, the STEP SDAI interface to access data and the database adaptor approach to persistence.

Chapter 4 begins by summarising the problems with current methods of CORBA data access solutions. From this a set of requirements is derived. Next, the high-level design of a Persistent Data Access to meet these requirements is described. The highlights of this design are:- a standard data definition model, the use of streams to cache data local to clients, the use of sessions to represent access to a datastore and integrate with the transaction and concurrency services.

Chapter 5 describes the design of the Stream Tunnel Service(STS). STS extends the OMG Externalisation service to permit the set-up and use of distributed streams. This service provides a way of transporting non-IDL defined data. STS encapsulates the actual network data transport mechanism used to transfer data between the ends of the stream, therefore allowing for efficient methods of data transfer than the ORB to be employed. An investigation is carried out into the performance of an ORB based stream and a connected socket based stream.

Chapter 6 begins by describing the Data Object Service(DObS). DObS is an abstract service for the management of persistent data. Management operations include identifying, creating, retrieving, storing and deleting persistent data. DObS is similar to POS in some aspects, but DObS most significant difference is the use of a single standard data transfer mechanism i.e. the Stream Tunnel Service. DObS is only abstract, it must be extended to handle specific types of data. The File Data Object Service(FileDObS) is one such extension allowing access and manipulation of files. An implementation of FileDObS is outlined that has a Java client side and a C++ server side. The chapter also presents the lessons that were learned from creating the STS and FileDObS applications.

Chapter 7 provides an outline of the Persistent Data Access Service(PDAS). PDAS is the decisive service to provide access to complex persistent data. PDAS embodies the high-level design features presented in chapter 4. PDAS consists of three modules:- client session, server session and the Entity Data Object Service(EntityDObS). EntityDObS extends DObS to enable the management of entities that make up the complex persistent data. EntityDObS fully provides a mechanism to cache data in the client session. The behaviour of components implementing the PDAS interfaces are demonstrated to show how PDAS carries out its task.

Chapter 8 briefly discusses the areas that PDAS can be used in. This includes how the Epicentre/DAE datastore can be supported by PDAS, how PDAS can act in a similar way to database adaptors and the requirements to access general datastores to provide clients with standard data access interface.

Chapter 9 reviews whether the aims and requirements of the research have been met, following this conclusions are presented.

1.7 Summary of Contribution

The work presented in this thesis results in the following main contributions for the area of CORBA and persistent data access integration:

- A framework of CORBA services to provide data access to heterogeneous datastores.
- The mechanisms necessary to support heterogeneous datastores i.e. a caching mechanism, a meta-schema model and data manipulation interfaces.
- The services to allow insulation of clients from the proprietary data access interfaces and access to already existing data in traditional datastores.
- How transaction and concurrency services can be integrated into services.
- Providing a copy-by-value mechanism to CORBA using streams.
- The use of more efficient network data transport mechanisms to transfer large amounts of data in CORBA.
- A distributed file service to Java applications that might be running in secure environments preventing local file access.
- The services are designed in a way which takes into account the nature of Epicentre data including complex networks of objects, large scientific arrays of data and generally the large number of object types in the Epicentre schema.

This research has resulted in the presentation of the paper – "Data Access and Transportation Services for the CORBA Environment", at a prestigious conference in the USA. A journal article has also been submitted to TAPOS (Theory & Practice of Object Systems).

Chapter 2

Introduction to the Background Standards and Technologies

2.1 Introduction

This chapter introduces the standards and technologies that are the basis for the problem domain. It provides an overview of the OMG CORBA architecture including the CORBA object models and the individual components of a CORBA implementation. The chapter progresses by outlining some of the OMG CORBA services. These explanations of CORBA services will be useful in understanding the solutions to the problem that has been set. Next, an overview of NIST's STEP international standard is described incorporating an introduction into the EXPRESS information modelling language. The final standard is POSC's SIP. SIP's major role is data exchange and management in the Exploration and Production industry. The various parts of the standard are presented with a further explanation of the Epicentre data model and the Data Access and Exchange interface. The chapter concludes with a discussion of the strengths and weaknesses of the standards.

2.2 The Object Management Group

The Object Management Group(OMG)[OMG 97] was founded in May 1989, by eight companies that wanted to promote the use of object-oriented technology by creating industry standards. Today, the consortium of companies making up the OMG has risen to over 800 member organisations. This demonstrates the importance and success that the OMG standards have had on the object software industry.

"The organization's(OMG) charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems."[OMG 95]

The first step the OMG took in realising its goal was the Object Management Architecture(OMA)[OMAG 90]. The OMA includes the Abstract Object model and the Reference model. The Object model provides standard definitions for the object-oriented model. The Reference model provides a model on how systems should be structured.

2.2.1 The OMA Abstract Object Model

The OMA Object model provides standard concepts, semantics and terminology for the object-oriented model, so that further standards and systems built using the OMA can have an unambiguous common model.

A summary of some of the basic definitions of the object model are as follows:-

- An object has an associated state and a set of operations.
- An object provides services to clients.
- Clients can invoke a service of an object by issuing requests.
- A request specifies a handle, an operation name and zero or more parameters.
- A handle identifies an object providing a service.
- An object performs some action as the result of a request.
- An object has a unique identity within the system that is identified by a handle.
- An interface describes the set of operations an object provides.
- An object must satisfy an interface i.e implement its defined operations.

- The code that performs a request is called a method.
- An implementation is the code supporting an object.
- For implementations classified as persistent, their state survives the process/thread.

2.2.2 The OMA Reference Model

The OMA Reference model(Figure 2.1) is a high level architectural view of how systems should be structured. The four elements of the Reference model are:-

- Object Services are a set of services made that provide basic level functionality to objects and applications e.g. instance management, storage management. These services should be common to each OMA platform, thus promoting reusability and portability of OMA applications.
- Common Facilities are application level services e.g. user interfaces, electronic mail. Common facility services are optional for OMA platforms.
- Application Objects are collections of objects that constitute an end-user application.
- Object Request Broker(ORB) is the mechanism allowing requests to be invoked on objects. Each of the other elements of the OMA use the ORB for interoperability.

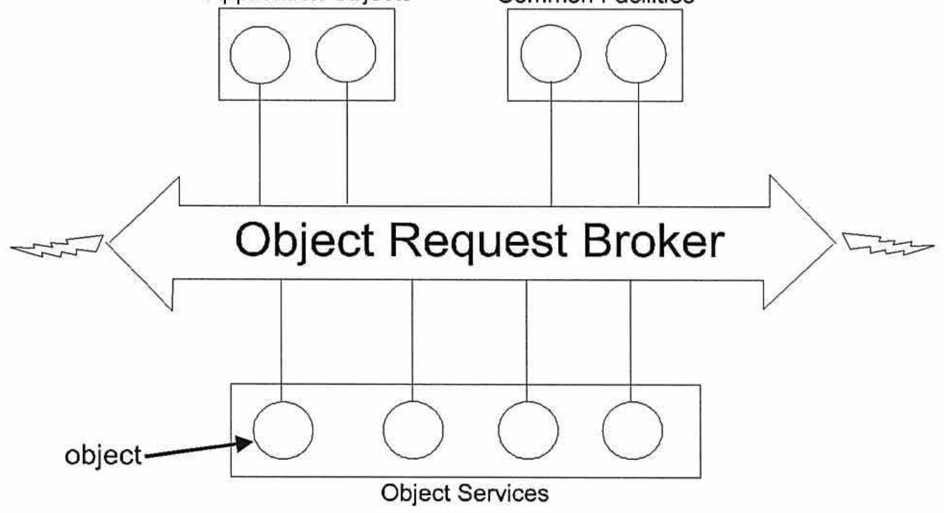


Figure 2.1: *Components of the OMA Reference Model*

2.3 The Common Object Request Broker Architecture

The Common Object Request Broker Architecture[OMG 95][Siegel 96][Orfali 96] is an OMG specification for the implementation of the OMA Object Request Broker(section 2.2.2). CORBA provides a standardised open communication middleware specification.

The Common Object Request Broker Architecture is commonly referred to by its acronym - CORBA. Implementations of CORBA give the ability of different parts of a system to easily inter-operate, regardless of the location, hardware, operating system and implementation language of the various parts.

The CORBA model[Kim 95] is based on the OMA Abstract Object Model(section 2.2.1), thus the CORBA model is of client objects/applications invoking a method of a server object. In the CORBA model, this process of invocation is called a request. The mechanism that transports requests between applications in a heterogeneous distributed environment is the Object Request Broker(ORB). To provide a service, a server object must have an explicitly defined interface. The interface describes the methods the object implements and the parameters passed and returned in the method. The interface is declared in the OMGs Interface Definition Language(IDL). It is the IDL descriptions of object interfaces that are critical to interoperability of CORBA over heterogeneous systems. IDL is the contract that both client and server will abide by. It is then a process of mapping IDL to the local programming language and system, so clients can make requests and servers can respond to requests.

The next sections describes the main features of CORBA in more detail and discusses areas of CORBA programming and systems that will be useful in understanding the interface frameworks and applications described in this thesis.

2.3.1 The Object Request Broker

The mechanism that conveys requests between clients and server objects is the Object Request Broker(ORB)(Figure 2.2). The ORB is a layer of software that encapsulates the network environment and hardware/software differences between the two ends of the request. Thus, the ORB effectively gives a client a single view of a heterogeneous distributed environment. This view is of numerous server objects, which the client can somehow gain references to, using the reference the client can request method invocations on the server objects through the ORB. This view is conceptually equivalent to the messaging mechanism between object instances within a single object-oriented language program.

Clients are not limited to just inter-operating with server objects on their native ORB. Requests can also be invoked on server objects present on remote ORBs. This ability is due to two standard CORBA inter-ORB communication protocols. The first protocol is the General Inter-ORB Protocol(GIOP). GIOP specifies a standard message and data format that ORBs can understand and use to interact with each other. The second protocol is the Internet Inter-ORB Protocol(IIOP). IIOP specifies how GIOP messages are exchanged over TCP/IP[Rieken 92] i.e. the internet.

IIOP is a valuable asset to the CORBA specification, as it allows the vast reach of the internet to be used as an ORB communication network. Currently, the most popular tool for providing interactivity between a user and an internet server is the Common Gateway Interface(CGI)[Orfali 97] over Hypertext Transfer Protocol(HTTP). Nevertheless, CGI/HTTP is an extremely slow and clumsy middleware mechanism to use compared to CORBA IIOP. For these reasons, the most popular web browsers can come with a Java ORB built in.

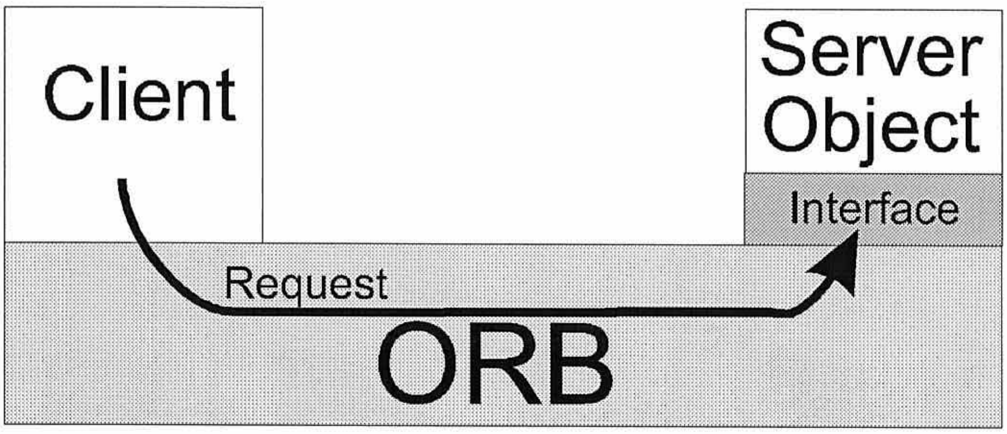


Figure 2.2: *Client requests operation of a server object through the ORB*

2.3.2 The OMG Interface Definition Language

To publish the methods a server object implements, the server object has an explicitly defined interface. Each interface is described in the OMG's Interface Definition Language(IDL)[OMG 95][Vinoski 96]. An IDL description presents a complete description of a server object that is relevant to a client for interoperability. Objects that implement an IDL interface are termed CORBA objects, and as a CORBA object is accessible from an ORB, it is also a distributed object. A more generic name for a CORBA object is a component, due it providing some known functionality and a public interface for making use of its functionality.

IDL can be used to specify the following of CORBA interfaces :-

- Interface name.
- Operations of the interface.
- Parameters of the operations.
- Semantics concerning the direction of use of operation parameters i.e. in, out, inout.
- Exceptions an operation can raise.
- Interface attribute variables.
- Type definitions including simple typedefs, sequence definitions, structures.
- Interface inheritance.
- Module definition to group interfaces in a common name space.

Figure 2.3 shows a sample IDL file showing some aspects of the IDL features mentioned above. IDL is independent of any programming language, but as the sample shows, IDL has a similar appearance to C++[Strou 91] class declarations.

IDL is the key feature of CORBA that enables CORBA systems to span heterogeneous hardware and software environments. This is due to code that is native to each environment being produced from the IDL definition. The codes main purpose is to translate between the standard data format of the ORB and the environment native data format i.e. marshalling and de-marshalling of data[Coulouris 88].

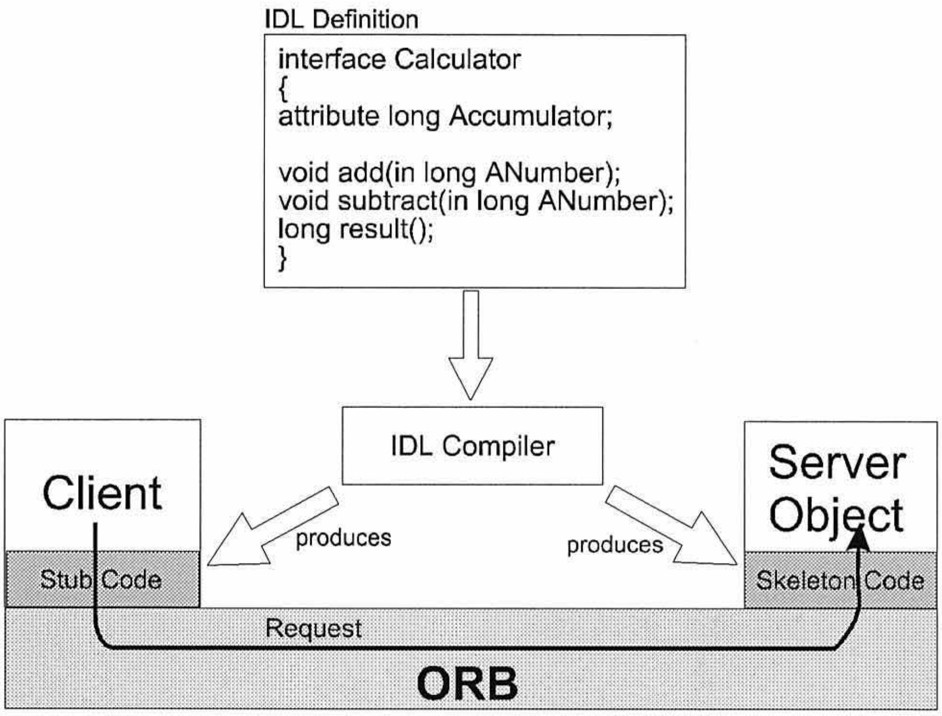


Figure 2.3: *Example IDL code and its produced native stub and skeletion code*

The native code produced from the IDL definition comes in two parts, these are the stub code and the skeleton code. Figure 2.3 shows the process of parsing an IDL file. The stub code is linked with the client. The stub code contains proxy classes that represent the server interfaces locally. To initiate a request, the client invokes the desired method of the relevant proxy class instance. The proxy object performs any necessary marshalling of parameter data and then makes a call to the ORB to carry out the request.

The skeleton code allows an invocation from the ORB to the implementation object of the IDL interface. Implementations of interfaces have to be bound to the skeleton code, so that the skeleton code can invoke an implementation method in response to a request.

For clients to invoke a request or for a server object to process a request in a program, the program must keep to a certain set of rules on the usage of the ORB. These rules are specific to each programming language and are termed the language bindings. Some areas of use that language bindings enforce are :-

- Use of IDL data types
- Use of object references

- How requests are invoked
- How parameters are passed and returned
- Whose responsibility is the allocation and deletion of parameters, whether the client, ORB or server.
- Use of exceptions including their raising and catching.

Programs must follow their language bindings to be CORBA compliant, and also to work in harmony with the ORB and its own stub and skeleton code.

Currently, in CORBA 2.2 there are six specified language bindings these are C, C++, SmallTalk, COBOL, Ada and Java. Note that CORBA applications do not necessarily have to be created from an Object-Oriented(OO) language. Procedural languages can also be used. However the mapping to CORBA is not as natural as with an OO language.

2.3.3 Object Adaptors

The object adaptor sits between the ORB and the server skeleton. Server objects have to register their presence with the object adaptor. There can be many server objects per adaptor, therefore, the adaptors main purpose is to accept requests from the ORB and route them to the targeted server object.

The Basic Object Adaptor(BOA)[OMG 95][Orfali 96] is CORBA's standard object adaptor, and a CORBA compliant ORB must support it. The Basic Object Adaptor provides all the basic facilities just mentioned. Additional facilities can be incorporated in an object adaptor, such as persistence and dynamic loading of objects[Baker 97].

Each object adaptor will have an activation policy. The activation policy defines how an object will be activated to carry out a request. CORBA defines four activation policies for the Basic Object Adaptor. These are :-

- Shared server - all server objects share the same process and one request is processed at a time.
- Unshared server - a new process is started for each active object.
- Server-per-method - a new process is started for each request.

- Persistent Server - a new process is started by a means outside of the BOA.

The most common activation policy is the shared server.

2.3.4 Object References

To make a request, a client must have an object reference to the CORBA/server object. An object reference is represented as some language specific encapsulated construct e.g. a C++ object. The object reference will contain enough information for the ORB to locate the CORBA object and make an invocation.

CORBA has an object-oriented architecture and a CORBA object reflects this by being the implementation of a hierarchy of IDL interfaces. Object references to CORBA objects can accordingly be cast up or down their hierarchy of interfaces.

A useful feature of CORBA object references is that they can be translated to and from a string format. This means that the location information contained in the object reference can be coded into a string of characters, and at some later time, the object reference can be recreated from the information in the string. This feature can be very useful, for example to store an object reference persistently in a file.

The location information embedded in object references is hidden from applications, and cannot be created or edited. In a CORBA compliant application the only element allowed to create object references is the object adaptor. An object reference is only produced when a CORBA object registers itself with its object adaptor. It is then the responsibility of the object to make itself known to potential clients by somehow passing an object reference to them.

There are a few common ways a client can gain an object reference to a required CORBA object :-

- From another CORBA object : the object reference is passed as a parameter in a request.

- Via the Naming service : the object reference is registered with label in the naming service.
- From a well known file : The client can read the object reference in string form from a file that the client knows the location of. This file could be located in a local or shared file system.
- From the ORB : the method 'resolve_initial_references' with a service name as a parameter can be called on the ORB. The ORB will return an object reference for the object server. This is the usual manner of gaining an object reference to the Naming Service. However, registering an object server with an ORB is highly ORB implementation specific and is an ORB configuration process rather than a dynamic run-time operation. Thus, it should only be used for a few well known and used services such as the Naming service.
- Via the Trading service : the Trading service is more a search oriented service for gaining object references than the Naming service. To be found in the Trading service, a server object registers properties that describes the service it offers. The client can then find a server object by searching for a specific set of properties.

Object references are proprietary to a specific ORB implementation i.e. they cannot be passed to another vendor's ORB implementation and utilised. This would be a major drawback for CORBA if this was the only situation, as different ORBs could not inter-operate. Fortunately, the CORBA specification for IIOP defines a standard format for an object reference called the IIOP Inter-operable Object Reference(IOR). The IIOP IOR contains the important information needed to locate server objects on TCP/IP network, such as the host's IP address[Stevens 90], the port number where a server is listening and an object key that indicates the target object. The IIOP IOR is a very beneficial element to ORBs that support the IIOP protocol, as they can simply pass IIOP IOR between each other and use them to request methods of server objects on other ORBs.

2.3.5 CORBA Object Summary

CORBA Objects are the principal processing element of CORBA, that together create a distributed CORBA application. Therefore, it is important to have a clear view of what a CORBA object is.

A CORBA object comprises an IDL interface and programming language code that implements the interface. The interface declares the operations and inherited interfaces of the CORBA object. The purpose of a CORBA object is to act as a server object by providing its operations for invocation to the distributed environment. Clients can request operations of the CORBA object through the ORB.

To make a request on a CORBA object, the client must have an object reference to it. Object references are created when the CORBA object registers with an object adaptor. It is the responsibility of the object to make its object reference known to potential clients.

Clients of a CORBA object can be applications or other CORBA objects. Consequently, a framework of interacting CORBA objects can be built. Together, the framework of CORBA objects will provide a higher level of service, which can be reused to supply some basic functionality to applications.

The key feature of CORBA is that this interaction between CORBA objects is independent of language, location and platform of the implemented object.

2.3.6 Further Features of a CORBA ORB

Figure 2.4 shows some further standard features of a CORBA ORB. These additional features will be briefly discussed to provide a fuller view of CORBA.

The Interface Repository[OMG 95] is a facility that stores IDL interface definitions. These definitions can be queried at run-time, so that clients can find the structure of interfaces e.g. the interface hierarchy, an interface operations, operation parameters etc. The main use of the interface repository is to allow dynamic invocations[Siegel 96].

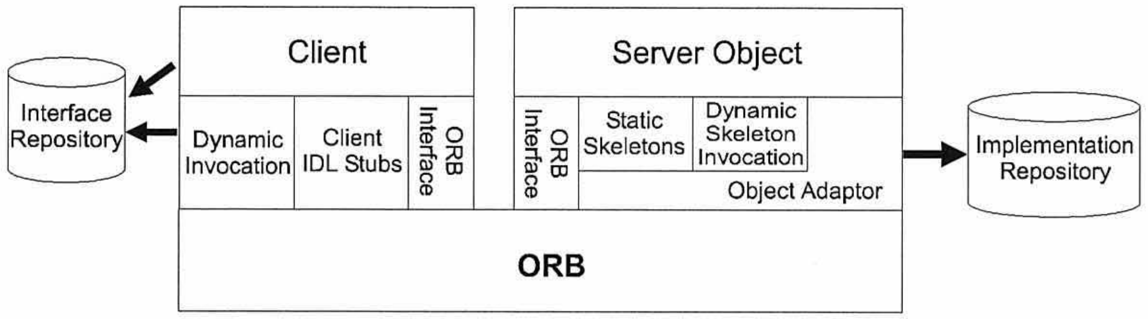


Figure 2.4: *Other major components of a CORBA ORB*

The Dynamic Invocation Interface permits clients to build a request for invocation on a server interface, even if the client has not been compiled with the stub code for the server interface. This is achieved by the client querying the interface repository to discover the structure of the desired interface method. Using this information the client dynamically builds a parameter list for the request and then performs the invocation.

The ORB Interface provides some useful functions that applications might require, such as converting object references to and from strings, finding initial services and partly used to make dynamic invocations.

The Dynamic Skeleton Interface[Baker 97] allows a server to receive requests for any defined interface, even though the server does not implement any interface. The primary purpose for this feature is to provide gateways to ORBs that use some other protocol for communication. The server can receive requests, format them to correspond to the other ORBs protocol and dispatch the request to the other ORB.

The Implementation repository provides run-time information on the interfaces a server process supports and the objects that are instantiated.

2.4 CORBA Services

The CORBA services[OMG COSS] are a set of OMG specified services supplying specifications for the implementation of the OMA object services. Each CORBA service provides some basic level functionality to CORBA objects and applications in the CORBA environment such as persistence, instance management and transactions. Each CORBA service specification is comprised of a set IDL interface describing the interface that clients interact with, and a description of the use and behaviour of the service.

One of the key design principals of the CORBA services is the so called Bauhaus principle:- *"Minimize duplication of functionality. Functionality should belong to the most appropriate service. Each service should build on previous services when appropriate."* [OMG 95b]

This principle is followed in the CORBA services with services reusing the functionality of another service. This principle has also been followed in the design of the services presented in this thesis.

The following sections introduce some existing CORBA services that have been used and built on by the new services presented in this thesis. All services can be found in [OMG COSS] which can be obtained from the OMG web site, further explanations can be found in [Orfali 96][Siegel 96][Baker 97]. The IDL modules for the services that are relevant to this thesis are shown appendix B.

2.4.1 The Naming Service

Any distributed system must have some mechanism for members of the system to find services that they require[Sloman 87]. Typically, this is achieved by a naming facility that stores location independent names and location dependent addresses for services. A member wanting to locate a service can look through the stored names for the name of the service it requires and, once found, can gain the address for the service. This scenario assumes that the member can initially locate the naming facility and that the member knows

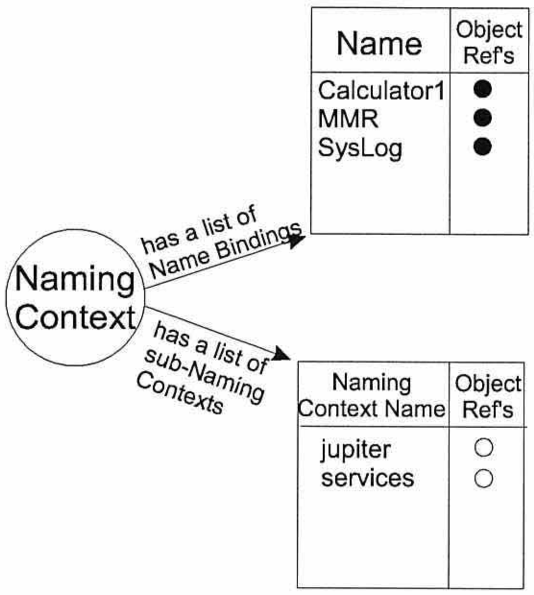


Figure 2.5: *Structure of Naming Context*

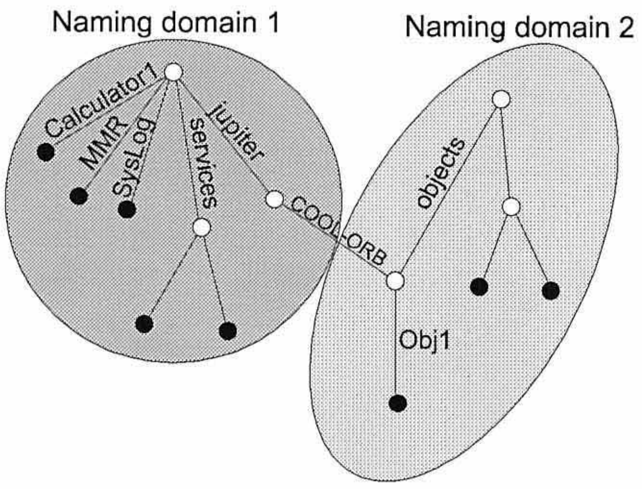


Figure 2.6: *Example of Naming Context graph*

the name of the service its looking for. As a naming facility is of utmost importance for a distributed system, the Naming service is one of the core CORBA services, and the most widely implemented service.

The Naming service stores string name and object reference pairs. Each pair is called a name binding(see Figure 2.5). A name binding must be explicitly created for a CORBA object to be registered in the naming service. A client can find a service by supplying a name, if a name binding exists for the name then the service returns an object reference to the client.

The service is not simply a one level list, but is modelled on a logical tree structure (Figure 2.6). Each node of the tree is called a naming context. Each naming context contains a grouping of name bindings and sub-naming contexts, and is analogous to a directory that contains files and sub-directories.

A naming context has an interface, hence it is also a CORBA object. The NamingContext interface allows for the creation and deletion of bindings and sub-NamingContext objects within the naming context. The interface also permits names to be resolved i.e. given a name, the naming context will return an object reference.

The actual name of an object is a sequence of strings that reflects the object position in the hierarchy of naming contexts. The fully qualified name of an object is the sequence of names between the root naming context and the object e.g. in figure 2.6, the name of obj1 is jupiter;COOL-ORB;obj1. Given a naming context other than the root, then the objects name is relative to the path from that naming context e.g. objects;obj1.

Another important use of the Naming service is a mechanism to allow the sharing of CORBA objects across separate ORBs. Naming services resident in separate ORB domains can be interconnected, thus the name bindings of server objects are accessible across the domains. The interconnection is achieved by binding a naming context resident in one ORB domain as a sub-naming context in the other ORB domain. This is assuming that the two ORBs can interoperate using a common protocol such as IIOP (see section 2.3.1).

2.4.2 The LifeCycle Service

The LifeCycle service is a specification for the management of CORBA object instances. The service provides interfaces and guidelines on how objects should be created, moved, copied and deleted. For an object to be managed by the LifeCycle service it must inherit and implement the LifeCycleObject interface. This interface provides operations to move, copy or delete the object.

To create objects, the LifeCycle service presents the concept of Factory objects. A

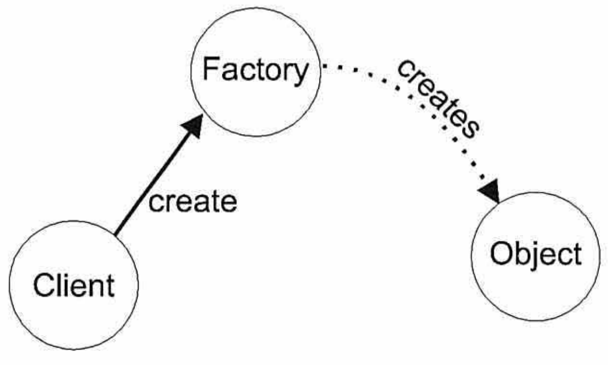


Figure 2.7: *Model of creating an object with the LifeCycle service*

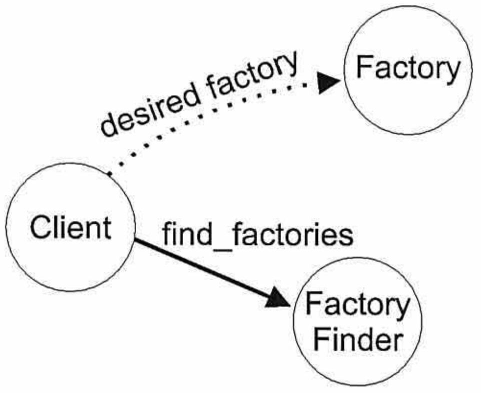


Figure 2.8: *Model of finding factory with the LifeCycle service*

Factory object is an object capable of creating another object (Figure 2.7). A client requests the Factory object to create objects for it.

Another important strategy the LifeCycle service suggests is how Factory objects are found. The service specifies the FactoryFinder object to achieve this. A client requests a FactoryFinder(Figure 2.8) to *find_factories* and passes some key information about the type of factory required. The FactoryFinder passes back to the client a list of factories that correspond to the key information. Using this model allows a loose coupling between client and Factory.

2.4.3 The Externalization Service

The Externalization service provides a framework of interfaces, to allow an object to externalise and internalise its state to and from a stream of data. Thus, an object can write its state to a stream (externalising), or an object can be recreated, by creating an uninitialised object and initialising its state by reading from the stream (internalising). The Externalization service framework comprises three primary interfaces, these are Streamable, Stream and StreamIO.

The Streamable interface has operations *externalize_to_stream* and *internalize_from_stream*. An object supporting these operations can be driven to write or read its state to or from the stream.

The Stream interface represents the data stream that Streamable objects write to and read from. The interface has two operations *externalize* and *internalize*. The *externalize* operation references a Streamable object for externalising. The *internalize* operation references a FactoryFinder object that is used to find a factory object. The factory is used to create an object that can initialise its state by internalising from the stream.

The StreamIO interface offers input/output operations to access data within a stream. These operations are of the form *write_<type>* and *read_<type>*, where *<type>* is any

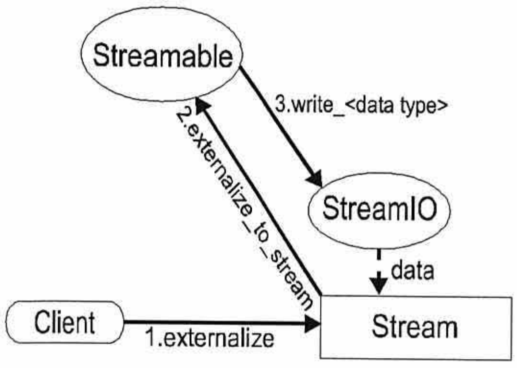


Figure 2.9: *Externalising an object to a stream*

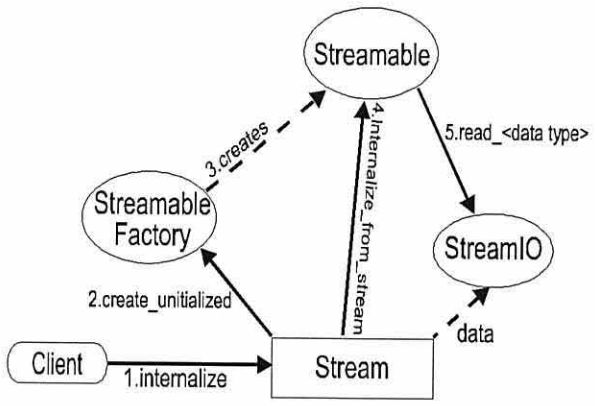


Figure 2.10: *Internalising an object from a stream*

basic type such as string, long, float or even a reference to another object to be externalised or internalised.

Figures 2.9 and 2.10 shows the sequence of requests necessary for an object implementing the Streamable interface to be externalised and internalised to and from a stream. The requests to externalise are:-

1. A client requests a Stream object to *externalize* a Streamable object that is referenced.
2. The Stream object requests the Streamable object to *externalize_to_stream* and references a StreamIO object.
3. The Streamable writes its state to the stream using the StreamIO write operations.

The requests to internalise are:-

1. A client requests a Stream object to internalize, a reference to a FactoryFinder is passed.
2. The Stream requests the FactoryFinder to find a factory object for creating the object contained in the stream.
3. The Stream requests the StreamableFactory to *create_uninitialized* that creates the new Streamable object.
4. The Stream requests the Streamable to *internalize_from_stream*.
5. The Streamable reads its state from the StreamIO object.

The Externalization service provides a standard method of getting state data into and out of a CORBA object and to/from a serialised form. This ability could be of great importance to CORBA applications, as it could be used to provide object persistence and object copy capabilities. These capabilities both require accessing a CORBA object's state and moving data in a serialised form. However, this requires the movement of data between designated points which the Externalization service does not provide.

A very simple form of persistence is already incorporated in the Externalization service, as the service allows streams to be created that save their data to files. However, the only control an application has over the objects persistent state is a simple string file name. There are no capabilities for locking, transactions, remote access and deletion of the persistent state.

The Externalization service could support object copying capabilities. This could be achieved by externalising an object to a stream, transferring the stream to another location and internalising it there, however the services specification gives no indication on a standard way of accomplishing this.

2.4.4 The Persistent Object Service

The Persistent Object Service(POS)[OMG COSS][Session 96] provides the functionality for CORBA objects to store their state persistently. This means that a CORBA object can save and load the data representing its state to/from some type of stable storage e.g. a file or database. Thus, the persistent state of the object can exist independently of the life time of the CORBA object instance.

POS was designed to permit any type of storage technology to be used as a persistent store such as files, relational databases and object-oriented databases. Allowing traditional storage technologies as well as directly mapped technologies (i.e. object-oriented databases) to be used for storage has the benefit of enabling access to already existing datastores. These existing datastores contain the vast majority of data for the world's computer systems. Consequently, they are a valuable asset to provide access to.

The diversity of POS is also one of its failures. To handle data in many different types of datastore, the service was designed very generically. This left the service under-specified, with implementors having to devise parts of the service themselves to compensate. Implementors have also found some fundamental flaws in the basic design of POS. As a result of these problems the OMG has withdrawn POS as a standard and is in the process of creating a replacement called the Persistent State Service.

An in-depth examination of the POS architecture, its benefits and failures are covered in chapter 3.

2.4.5 The Object Transaction Service

The Object Transaction Service(OTS) is probably the most significant service that will enable CORBA to be successful in being the infrastructure for business enterprise systems. OTS provides CORBA applications with the ability to perform requests within the context of a transaction[Bernstein 97].

"A transaction is a unit of work that has the following (ACID) characteristics :-
A transaction is atomic; if interrupted by failure, all effects are undone (rolled back).
A transaction produces consistent results; the effects of a transaction preserve invariant properties.
A transaction is isolated; its intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they performed concurrently.
A transaction is durable; the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure)."[OMG COSS]

The ability to perform operations within a transaction is critical for systems that must maintain the integrity of data and system state despite system failures. In these systems, all changes performed within a transaction must be made permanent i.e. commit, or in case of failures all changes must be cancelled i.e. rollback.

However, to allow operations to be performed within a transaction in a distributed environment is a complex problem. A part of this problem is that each resource, e.g. a

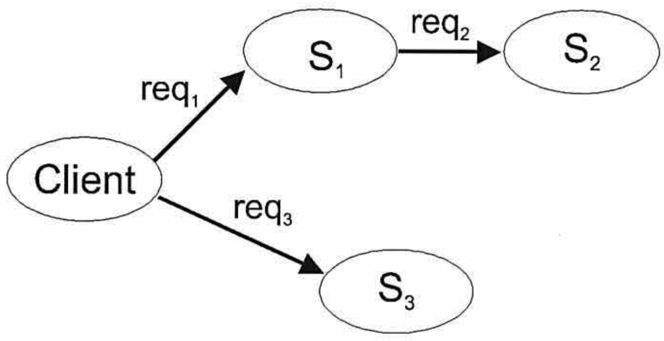


Figure 2.11: Ob*ject requests within a transaction*

server object, that performs some operation within the boundaries of a transaction must obey the decision of the transaction to commit or rollback.

For example, Figure 2.11 shows a number of requests(req_i) between a client and a number of CORBA objects(S_i) within a transaction that the client has started. The client invokes req_1 on S_1. In processing req_1, S_1 invokes req_2 on S_2, and the client then invokes req_3 on S_3. All objects: S_1, S_2 and S_3 must take part in the commit or rollback of the transaction and somehow the transaction must know of each object taking part.

The subsequent problem is to allow each resource to commit or rollback consistently. For example, the client wishes to commit, so the transaction sends commit signals to each server object sequentially. Firstly, S_1 commits successfully, then the commit signal is sent to S_2. If S_2 or S_3 could not commit and forced a rollback, then the transaction would not be atomic as S_1 has made its changes permanent. The solution is a two-phase commit[Bernstein 97][Coulouris 88] instead of a single phase commit.

The two-phase commit has a coordinator that is some entity driving the transactions commit. In the first phase, the coordinator sends each resource a prepare signal. The prepare signal is fundamentally asking the resource whether the resource can commit, the resource returns a value indicating whether it will commit or wants to rollback, this is called its vote. Each resource responds with its vote. If each vote is a commit, then in the second phase, the coordinator sends the commit signal to each resource and each resource makes its changes permanent in response. If one or more votes were to rollback, then the coordinator signals each resource to rollback.

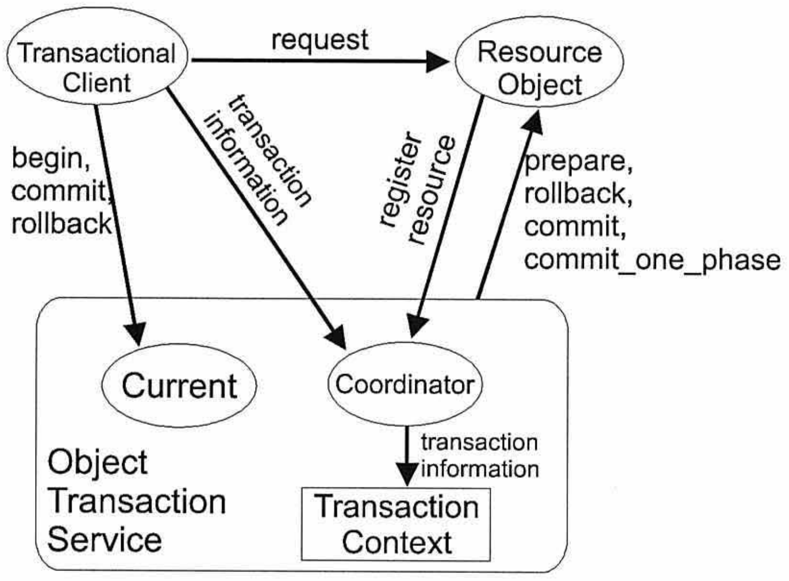


Figure 2.12: *Primary interfaces and components involved in a OTS transaction*

The Object Transaction Service(OTS) is a framework of interfaces defining a model of the behaviour of objects taking part in a transaction. The OTS model incorporates distributed transactions with a one- or two-phase commit protocol. Figure 2.12 shows a simplified overview of the primary OTS interfaces and components involved in a transaction.

The OTS entities are :-

- The Transactional Client is the entity that creates, starts and ends the transaction.
- The Recoverable Server is an entity that has recoverable state i.e. it stores data on stable storage, thus can recover from failures.
- The Object Transaction Service is a global system service that coordinates transaction and drives commit protocols.
- The Transaction Context is an entity that is copied to each server involved in a transaction. The Transaction Context contains information on the state of a transaction.

The interfaces are :

- Current interface provides begin, commit and rollback operations as well as access to the Coordinator object of the OTS.
- Coordinator interface provides operations to get information on the transaction and allows resources to register as taking part in the transaction.

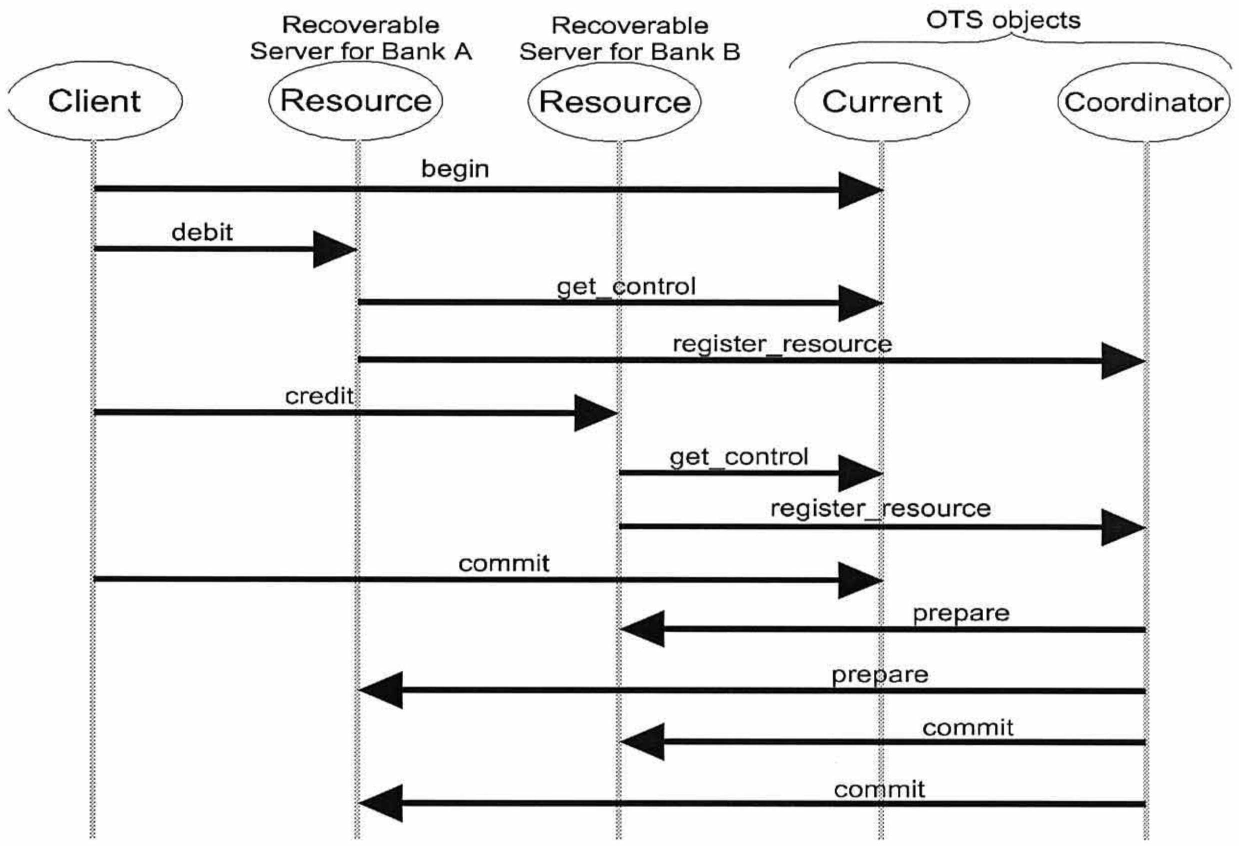


Figure 2.13: *Sequence of requests for a client to move money between two Bank objects in a OTS transaction using two-phase commit*

- Resource interface must be supported by recoverable servers, so that the OTS can drive a commit protocol with the server.

Figure 2.13 shows an example of an OTS scenario. This scenario is taken from [Orfali 96] and gives a clear view of the sequence of requests between objects within an OTS transaction, consequently it should give a better understanding of OTS. The example scenario shows a client starting a transaction, preforming a debit and a credit operation between two Bank objects and then committing the transaction, at which point the OTS performs a two phase commit. In the diagram, the horizontal arrows represent requests, the vertical lines represent the events (requests) that an object has received and initiated over the duration of the transaction.

1. The client begins the transaction.
2. The client debits Bank A.
3. Bank A implements recoverable behaviour, so it registers its Resource interface with the Coordinator.

4. The client credits Bank B.
5. Same as step 3 for Bank B.
6. The client issues a commit.
7. The Coordinator performs the first phase of the two-phase commit by requesting each Resource to prepare. Each Resource in turn returns a vote on whether to commit or not.
8. Each vote was to commit, hence the Coordinator requests each Resource to commit and the transaction is complete.

An overview of the fundamental model of the Object Transaction Service has been given. The OTS has more aspects such as nested transactions, recovery, transaction context propagation and supporting XA-compliant resources of the X/Open Distributed Transaction Processing model. These features are covered in the OTS specification[OMG COSS] and [Fleming 97], information on implementing an OTS can be found in [Grasso 97] and [Grasso 97b].

The Object Transaction Service is a complex and important CORBA service and is necessary for any CORBA system that has to be reliable. Thus, a CORBA persistent data facility serving a reliable system must be able to work within the context of a transaction and support the transaction ACID properties. Consequently, a persistent data facility must comply with the OTS to be of value.

2.4.6 The Concurrency Control Service

The Concurrency Control Service(CCS) manages access to shared resources(i.e. a CORBA object), so that concurrent accesses to the resource by multiple clients does not upset the consistency of the resource. To keep the consistency of the resource, the CCS enforces a locking mechanism[Coulouris 88] on access to the resource, whereby a client must gain a lock on a resource to access it. The two basic lock types of the CCS are read-only lock and write lock, allowing shared non-update access or exclusive updating of the resource's data. Typically, resources will implement a locking policy of multiple readers and a single writer. Thus, multiple clients can concurrently read from the resource, but only one client can update the resource with no other locks active on the resource.

The Concurrency Control Service(CCS) consists of a set of interfaces which a resource must implement to control access to itself with a locking mechanism. The primary interface is the LockSet interface, which provides operations for clients to explicitly gain and release locks on the resource.

The Concurrency Control Service is a complementary service to the Object Transaction Service(OTS) as CCS provides a locking facility that serialises access to a resource, thus providing the isolation characteristic of a transaction. For this reason, CCS has been designed to work with the Object Transaction Service by allowing the OTS to drive the release of locks.

Again, in a reliable system, the consistency of the system must be maintained and the concurrency control service is the manner of achieving this in CORBA systems. Hence, CCS is a key part of any persistent data facility.

2.4.7 The Relationship Service

In the CORBA environment, an object can refer to another object by holding an object reference to it, thus a relationship is formed between the two objects. In this model, the relationship details are hidden within the objects involved. Consequently, the relationship is not generically viewable by other objects and services that might wish to know about an objects relationships. The Relationship Service provides a means by which relationships can be created external to objects, hence relationships can be viewed and traversed by the objects involved in the relationship or by external objects using standard interfaces.

The two primary interfaces presenting a relationship are the Role and Relationship interfaces(Figure 2.14). The Role interface represents one end of a relationship. The interface has operations to traverse the relationship and to create or destroy a relationship. The Relationship interface represents the link between roles and has an attribute defining the relationship name.

The Role and Relationship interfaces define the basic level of functionality that the Relationship service supplies. The next level of the Relationship service allows the creation of complex relationship graphs. This level introduces a Node interface that permits an object to take part in multiple roles. At this level, a complex graph of objects and relationships can be created. The TraversalCriteria interface gives the ability to define a set of rules that define a sub-group of objects of a graph. The third level allows the specialisation of roles, by defining whether the role type is a containment or reference role.

The Relationship service can be used by other services to manipulate a graph of objects as a group. For example, the LifeCycle service can copy a group of objects from one place to another, and the Externalization service can externalise a group of objects to a stream. Both examples are dependent upon a set of rules that comprises the traversal criteria, which defines the group of objects.

The CORBA architecture is based upon components dynamically inter-operating by requesting operations of each other. The next sections introduce the STEP and POSC SIP architectures where inter-operability is based on applications sharing repositories of data with a defined data models. These data repository architectures are introduced to show the differences between them and the CORBA style of inter-operability. They also serve to provide an overview of the EXPRESS information modelling language that was created by the STEP project and to describe the nature of the Epicentre data model which is defined in EXPRESS.

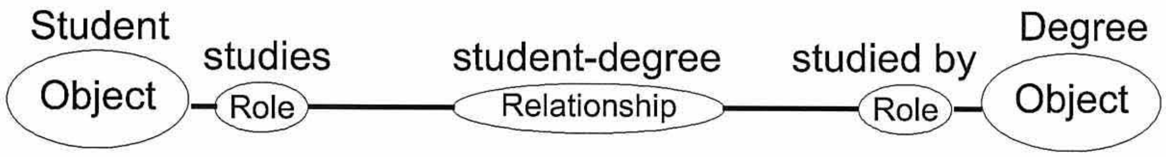


Figure 2.14: *Example of a relationship between objects using the Relationship service*

2.5 Overview of STEP

The **ST**andard for the **E**xchange of **P**roduct Model Data(STEP)[STEP 94][Owen 93] is the unofficial name for the international standard - *ISO 10303 Industrial automation systems - Product data representation and exchange*. STEP is under the guidance of the National Institute of Standards and Technology(NIST). The STEP project is working towards providing a single international standard for the exchange of product data. Product data is typically created and required by Computer-Aided Design(CAD), Computer-Aided Engineering(CAE) and Computer-Aided Manufacturing(CAM) systems throughout the life of a product. Therefore, it is important that a standard data exchange format is available to transfer product data between such systems.

2.5.1 The STEP architecture

STEP is realising its goal in three main ways, these are :-

- To provide a method of describing the structure of data independent of data storage.
- To define standards to allow the exchange and sharing of data.
- To provide standard data models for specific industry areas.

These activities are described in more detail in the STEP architecture.

The STEP architecture(Figure 2.15)[Fowler 96][Yang 95] is comprised of a number of sections. Each section of the architecture has a set of parts. Each part is an individual specification and has a number identifying the part. This number is relevant to the section number and part number.

The STEP architecture sections are :-

1. Description Methods

   Part 1: Overview and fundamental principles : defines the basic principles of STEP and the structure of the architecture overall.

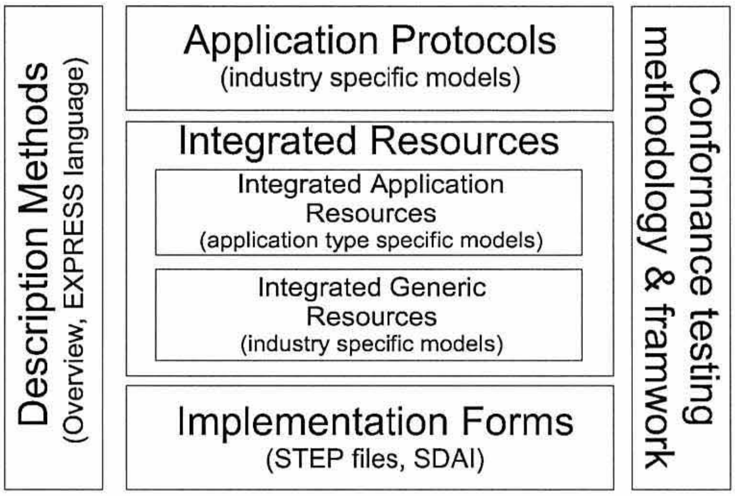

Description Methods
(Overview, EXPRESS language)
Application Protocols
(industry specific models)
Integrated Resources
Integrated Application Resources
(application type specific models)
Integrated Generic Resources
(industry specific models)
Implementation Forms
(STEP files, SDAI)
Conformance testing methodology & framwork

Figure 2.15: *The STEP architecture*

- EXPRESS : One of the key objectives of STEP is to provide an unambiguous, computer-interpretable representation of product data. To achieve this objective, the EXPRESS language[EXPRESS 92] was developed. EXPRESS is used to describe data in the form of entities that have attributes. Entities are linked together by relationships or inheritance. EXPRESS also provides a full procedural programming language to specify constraints on entity instances. The entity definitions for a single data model are grouped together within a named schema. These features make EXPRESS a powerful data modelling language that is independent of any data storage implementation.

2. Implementation forms

This section defines standard formats within which instance data belonging to EXPRESS defined models can be stored, exchanged and accessed. The first part of this section is part 21-the physical file format. Part 21 describes the format of how instance data can be stored in a file. These files are unofficially called STEP files and are the primary method of exchanging data between systems. Part 22 Standard Data Access Interface(SDAI) provides a standard interface to access data within an application e.g. a database management system, regardless of the internal form of the data storage. There are additional parts to SDAI that specifies a mapping of the SDAI to specific programming languages such as C, C++ and Java. There is also a mapping in development to the OMG's Interface Definition Language.

3. Integrated Resources : This section provides a set of common data model definitions that are required in many different product data application areas. This section is divided into two different parts:-
   - Integrated Generic Resources : This part provides data models that are independent of any specific application area, but is common to many different application areas. For example, Part 42 Geometric and topological representation defines generic representations for the shapes of objects.

   - Integrated Application Resources : This extends the integrated generic resources to support the needs of specific groups of applications. For example, Part 101 Draughting defines common data representations for all applications that make use of engineering drawings.

4. Application Protocols
   This section provides data models related to specific industries. The application protocols are data models that are more precisely defined than the more abstract models of the integrated application resources. It is these application protocols that are the pinnacle of the STEP architecture. Applications of a specific industry using their industry's application protocol, can easily exchange product data with similar applications.

5. Conformance testing methodology and framework
   This section provides information on methods for testing software product conformance.

2.5.2 EXPRESS Overview and Constructs

The EXPRESS information modelling language[EXPRESS 92] is an international standard(ISO 10303: Part 21) for the modelling of data independent of any implementation technology. EXPRESS is the key element to allow data exchange in the STEP architecture. Applications exchanging data can refer to a common EXPRESS-defined data model to understand the structure of the data.

The EXPRESS language syntax is precisely defined in a set of logical rules. Consequently, a data model correctly written in EXPRESS can be computer interpretable and an automated process can be employed to map the data model to any implementation technology e.g. a programming language or database management system.

EXPRESS embodies concepts such as units of information called entities that are comprised of attributes. The relationships and inheritance structure of entities can also be described. EXPRESS provides a procedural language to specify constraints on entity attributes and between entities. These concepts make EXPRESS more than a data definition language, EXPRESS is an information modelling language as it captures the structure and constraints of the data.

Figure 2.16 shows a sample EXPRESS file for a small hypothetical data model. The data model and EXPRESS code is neither a complete solution to such a problem or an example of good data modelling, but was simply created to demonstrate some of the fundamental constructs of the EXPRESS language.

Entity - Entities (e.g. ENTITY person, lines 10-19) are the basic units of data in EXPRESS. An entity is made up of a set of attributes and constraint rules on the values of these attributes. An entity can inherit the attributes and constraints of other entities by specifying the entity or entities that it is a sub type of. For example, line 22 specifies ENTITY student is a sub type of person.

Data Types - Each attribute of an entity has a data type. Data types can be simple types such as integer, real, string etc.. Data types can be more complex types such as defined types e.g. 'PCAS_code' defined on line 3 or they can be enumeration types e.g. 'degree_type' defined on line 6.

Relationships - a relationship is indicated by an entity attribute having a data type of another entity. For example, line 33, the degree entity has a one-to-many relationship with the student entity. The relationship is one-to-many as it is specifying an aggregate relationship with the 'SET OF' key words. However, this relationship definition only indicates the relationship in the direction of degree to student. The relationship in the

```
1.    SCHEMA engineering_school;
2.
3.    TYPE PCAS_code = STRING(15);
4.    END_TYPE;
5.
6.    TYPE degree_type =
7.    ENUMERATION OF (ee,cse,csb,mee,mcse);
8.    END_TYPE;
9.
10.   ENTITY person
11.   SUPERTYPE OF (student);
12.        surname   : STRING;
13.        first_name      : STRING;
14.        address   : STRING;
15.        age       : INTEGER;
16.        phone_no        : STRING;
17.   UNIQUE
18.     surname, first_name;
19.   END_ENTITY;
20.
21.   ENTITY student
22.   SUBTYPE OF (person);
23.                   code : PCAS_code;
24.     grade_average : REAL;
25.   UNIQUE code;
26.   INVERSE
27.     degree_course : degree FOR degree_students;
28.   WHERE
29.     age_limit : age <= 65 AND age >= 18;
30.   END_ENTITY;
31.
23.   ENTITY degree;
32.     degree_code : degree_type;
33.     degree_students : SET [0:100] OF student;
34.   END_ENTITY;
35.
36.   RULE maximum_number_of_masters_students FOR (student, degree);
37.   WHERE
38.     check : SIZEOF(QUERY(temp <* student |
39.          temp.degree_course.degree_code = mee OR
40.          temp.degree_course.degree_code = mcse)) <= 20;
41.   END_RULE;
42.
43.   END_SCHEMA;
```

Figure 2.16: *An Example EXPRESS schema*

opposite direction (i.e. student to degree) is specified on lines 28 & 29. The 'INVERSE' keyword indicates the relationship in the reverse definition.

Constraints - are divided into two categories, these are local and global rules. Local rules apply constraints on the value of attributes of an entity that the rules are defined in, or constraints on all instances of that entity. There are two forms of local rules : unique and domain. Global rules apply constraints amongst many entities in a data model.

Unique rules apply to all instances of an entity type. A unique rule specifies an attribute or set of attributes that must be unique across all instances of an entity. For example, lines 17 & 18 declare that the combination of 'surname' and 'first_name' must be unique for each person instance.

Domain rules constrain the value of an attribute for a particular instance. For example, lines 28 & 29 declare a rule that constrains the age of student.

Global rules are defined using the procedural language part of EXPRESS. The procedural language allows rules to be formed. These rules involve querying properties of instances of differing entity types. For example, lines 36-41 query all student instances to find out if their degree type is a masters degree. The rule limits the number of masters degrees to less then or equal to 20.

The EXPRESS language is an excellent data definition language to represent complex data models. This combined with the ability to define constraints makes EXPRESS a very powerful information modelling language. For these reasons and by virtue of being an international standard, EXPRESS has been widely used outside the STEP project, as well as in many other standards e.g. POSC SIP[POSC 92].

2.6 POSC's System Integration Platform(SIP)

The Petrotechnical Open Software Corporation(POSC) is an organisation formed by the Exploration and Production(E&P) industry i.e. the oil and gas industry. POSCs purpose is to produce standards for computing technologies within the E&P industry. The need for these standards was due to E&P companies in the past building expensive software applications. These applications could not communicate or exchange data with other applications as they were built using proprietary communication and data format standards.

The ability to share data between applications in the E&P industry is an important one, as most applications have one primary processing task and must pass data onto other applications to carry further processing. For example, in searching for oil, a seismic survey has to be carried out. The seismic recorder application places the seismic data onto magnetic tape. To view the data, a 3D seismic interpretation application must understand the stored format of the data to be able to read and interpret it.

POSC's solution to the problem of interoperability and data exchange in the E&P industry is the Software Integration Platform(SIP) architecture[POSC 92]. SIP is a set of specifications that not only standardises methods of sharing and exchanging E&P data, but also some aspects of general computing environments such as user interface styles and computing platforms.

2.6.1 The SIP specifications

SIP currently comprises of seven specifications, these are :-

1. Base Computer Standards - this specification standardises some aspects of computer systems such as operating systems and language compilers. These are specified to increase portability of SIP applications across systems.

2. Epicentre Data Model[POSC 95] - is a very large data model defined in EXPRESS[EXPRESS 92](section 2.5.2). The entity definitions in Epicentre can be used to describe the vast majority of data that will ever need to be stored in the E&P industry. Thus, Epicentre provides a common reference model between E&P applications.

3. Data Access and Exchange(DAE)[POSC 95b] - is a specification for an Application Programming Interface(API) to access Epicentre defined data. The API is independent of any underlying storage technology(e.g. DBMS). This is to isolate the standard from any technological changes in storage technologies e.g. migrating from relational databases to object-oriented databases. The DAE specification also incorporates its own variation of SQL. This permits developers to manipulate Epicentre data in a query type style.

4. POSC Exchange Format - enables the storage of Epicentre-defined data in a standard external form that can be exchanged with other systems.

5. User Interface Style Guide - specifies a common computer-human interface that defines the appearance and behaviour of user interfaces.

6. Computer Graphics Metafile Petroleum Industry Profile - defines formats for the exchange of graphical data.

7. Inter-Application Communications - provides a standard API for inter-operability between applications.

2.6.2 The Epicentre Logical Data Model

Epicentre[POSC 95][Kim 95] is a data model describing more than 1500 business and technical objects concerned with the E&P industry. These objects are termed entities in Epicentre. The definitions of each entities attributes and relationships are accurately represented in EXPRESS.

Epicentre is a logical model as it cannot be directly implemented in a physical database, hence is independent of any storage technology. However, POSC does provide a set of data definition language statements that provides a mapping of Epicentre to some popular relational databases. These mappings are called projections.

Each Epicentre entity has a complex structure due to the many attributes and relationship attributes. Additionally, each entity can inherit from multiple entities. This can make the inheritance hierarchy of an entity complicated and greatly increases the number of attributes for a given entity.

```
ENTITY well
 SUBTYPE OF (composite_spatial_object, facility, product_flow_network_unit);
 identifier : ndt_name ;
 ref_naming_system : OPTIONAL ref_naming_system ;
 ref_well_structure_rule : OPTIONAL ref_well_structure_rule ;
 part_of_well : OPTIONAL SET [0:?] OF well ;
 well_slot : OPTIONAL well_slot ;
INVERSE
 well_status : SET [0:?] OF well_status FOR well ;
 composed_of_well : SET [0:?] OF well FOR part_of_well ;
 wellbore : SET [0:?] OF wellbore FOR well ;
 well_report : SET [0:?] OF well_report FOR well ;
 well_alias : SET [0:?] OF well_alias FOR well ;
 well_surface_point : well_surface_point FOR well ;
 well_surface_feature_role : SET [0:?] OF well_surface_feature_role FOR well ;
 well_activity : SET [0:?] OF well_activity FOR well ;
 well_completion : SET [0:?] OF well_completion FOR well ;
 pty_economic_oil_cut_limit : SET [0:?] OF pty_economic_oil_cut_limit FOR well ;
 pty_specific_productivity_index : SET [0:?] OF pty_specific_productivity_index FOR well ;
 pty_economic_water_cut_limit : SET [0:?] OF pty_economic_water_cut_limit FOR well ;
 pty_unit_productivity_index : SET [0:?] OF pty_unit_productivity_index FOR well ;
 pty_economic_gor_limit : SET [0:?] OF pty_economic_gor_limit FOR well ;
 pty_economic_wor_limit : SET [0:?] OF pty_economic_wor_limit FOR well ;
 pty_productivity_index : SET [0:?] OF pty_productivity_index FOR well ;
 pty_economic_limit_money : SET [0:?] OF pty_economic_limit_money FOR well ;
 reservoir_drainage_feature : SET [0:?] OF reservoir_drainage_feature FOR well ;
UNIQUE
 si: identifier, ref_existence_kind;
END_ENTITY;
```

Figure 2.17: *EXPRESS code for the Epicentre Well entity*

Figure 2.17 shows the EXPRESS code for the well entity. The `well` entity has many relationship attributes with other entities, where many of these relationships are aggregate (zero-to-many) relationships. For example, a `well` can contain many well bores(i.e. drilled holes). This relationship is depicted in the *`wellbore`* attribute.

The code declares the attributes of only the well entity and does not include the definitions of the supertypes of `well`. Figure 2.18 shows the inheritance hierarchy of the `well` entity. The total number of attributes that the `well` entity possesses and inherits from its supertypes is 64.

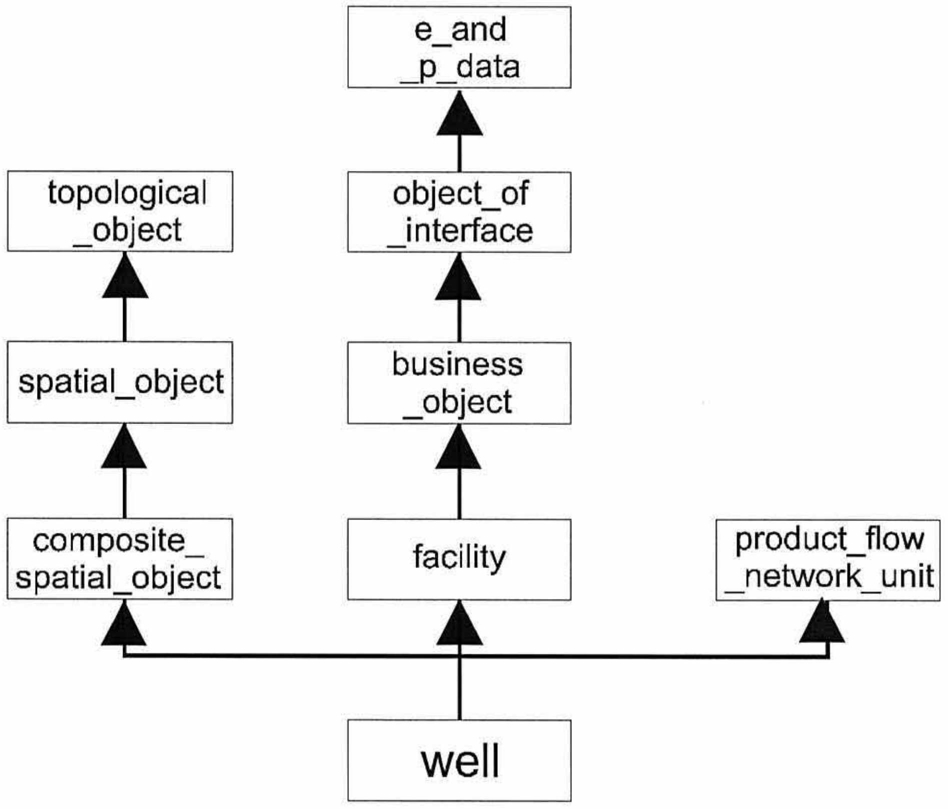


Figure 2.18: *Entity hierarchy of Well entity*

2.6.3 The Data Access and Exchange Specification

The Data Access and Exchange[POSC 95b] specification is an Application Programming Interface(API) to access Epicentre-defined data, including creating, querying, updating and deleting the data. The API also allows management of the data access environment such as transaction management.

An implementation of the DAE is termed a Data Access and Exchange Facility(DAEF)(Figure 2.19). An accessible DAEF storing Epicentre data is termed a POSC datastore. Applications written using the DAE API specification should be portable across DAEFs of different vendors.

The DAE API specification can be categorised into the following sections :-

- General environment operations - comprises of connection, transaction, error status and memory management operations.
- Instance operations - provides the direct manipulation operations, including creation/deletion of instances, attribute reading/updating and aggregate management.
- Frame operations - much of E&P data are large sets of scientific data or spatial data. These data sets are stored in arrays. The arrays may not only be large in size, but can also have many dimensions. These complex arrays are handled by the DAE in the form of frames. A frame holds an array of data or can hold an array of child frames. Using this feature recursively, a structure representing an array of many dimensions can be built up.
- Query language and execution - the Epicentre data can be directly manipulated using

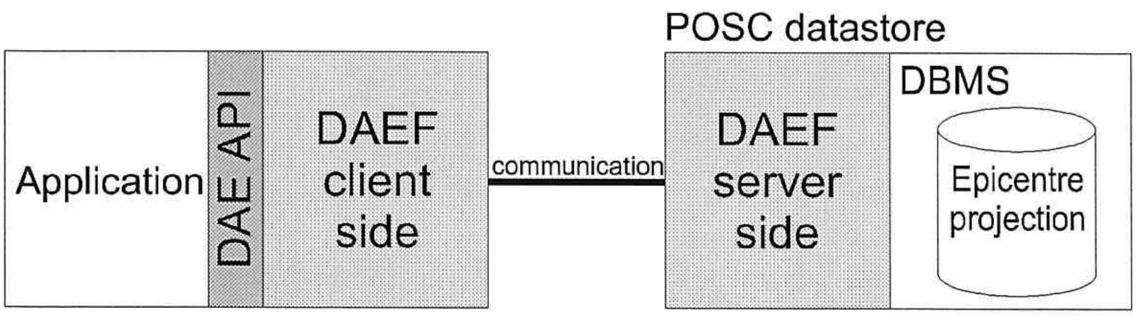


Figure 2.19: *Structure of a DAEF*

instance operations or can be manipulated in a query type style using the DAE data access language. The data access language is based on SQL. Its formal specification is provided and explained in this section. The execution of data access statements using DAE operations is also specified.

- Administration operations - provides control over user access, and creating or deleting datastores.

2.6.4 Example DAE Code

An example of DAE code is shown in figure 2.20. The code is shown to give a clearer view of what the DAE does and how it is used.

The DAE API is a set of 'C' functions that can be called to manipulate Epicentre instances resident in a POSC datastore. The example code demonstrates the two forms of entity manipulation using the data access language and direct manipulation. The data access language is used to execute a query to return a `well` instance with a specific ***`identifier`*** attribute. Direct manipulation is then used to change the ***`identifier`*** attribute.

The code execution starts by creating a connection with a specific DAEF server(6). This connection is called its session. A datastore is then opened with the server(8). The query to find a specific well(3-4) is then executed(10-11). The result of the query is fetched(13-15), and then a direct manipulation operation is used to change the value of the selected `Well` ***`identifier`*** attribute(19-20). The changes are committed(24), memory is released and the session is disconnected(26-29).

Note that the example code does not show any checking that a DAE function call has successfully executed. In a reality, each DAE function call would be individually checked for correct execution. DAE function calls that are particularly prone to failure are attribute update operations(19-20). This is due to constraints on attribute values specified in the data model. The ***`identifier`*** attribute of `well` has a unique constraint on it(see Figure 2.17). If this unique constraint is not satisfied then the update operation will return an error.

```
daeStatement stmtHandle = NULL;
daeInstance wellInstance = NULL;
daeString queryStatement
  = "SELECT well FROM well WHERE identifier = 'Alpha00001'";

daeConnectSession(user, password, NULL, "myDAEFserver", &session);

daeOpenDataStore(session, "datastore1", &datastore);

daeAllocStmt(datastore, &stmtHandle) /* allocate the query statement */
daeExecDirect(stmtHandle,  queryStatement, -1); /* execute the query */

daeBindCol(stmtHandle, 1, DAE_C_INSTANCE, &wellInstance, NULL,
                      &indicator);  /* bind results of query to wellInstance
             */
daeFetch(stmtHandle,&returnCount); /* fetch the first result */

if (returnCount!=0) /* if any instances are returned */
{
 daeUpdateValueOfAttribute(wellInstance,"IDENTIFIER",/*update identifier*/
                          DAE_C_STRING, "Beta00001") /*attribute of well*/
}
else printf("\n Query did not return any wells");

daeCommitTransaction(session); /* commit the changes */

daeFreeHandle(wellInstance));
daeFreeHandle(stmtHandle);
daeFreeHandle(datastore);
daeDisconnectSession(session); /* disconnect the session */
```

Figure 2.20: *Example DAE 'C' code*

2.7 Summary and Conclusions

This chapter has summarised three major international computing standards. Each standard tries to achieve integration of heterogeneous applications by providing inter-operability models and common data models. However, the standards place a different emphasis on the importance of the issues of inter-operability and common data models. Figure 2.21 shows a synopsis of the primary parts of the three standards, the parts are ordered vertically according to similar functionality.

The STEP and POSC SIP standards are greatly oriented towards providing a common data model that is used as a template to create data. Inter-operability between applications is in two forms - exchange and sharing of data.

Data exchange(Figure 2.22) is carried out by an application storing the data in a standard format in a file e.g. Part 21, PEF. Another application can read the stored data by referencing the common data model and by complying with the standard format rules.

Data sharing(Figure 2.23) entails applications using a standard interface(e.g. DAE, SDAI) to concurrently access data in a shared datastore. Applications using the data access interface have to conform to a common logical data model(e.g. Part 218 Ship Structures, Epicentre) that the datastore supports. It is the duty of the software layer implementing the data access interface to do the conversion between the data storage model and the logical data model.

Both STEP and SIP have a similar architecture to inter-operability by having a common data model(s) and instance data of the model can be exchanged or shared. The difference is that SIP has a single data model supporting the needs of a single industry, while STEP has a many data models supporting a wide range of industries. Due to SIP's focus on a single model, its data exchange and sharing standards are closely tied to the data model i.e. Epicentre, while STEP standards are more oriented to sustaining multiple models.

CORBA provides inter-operability by supplying a communication system(i.e the ORB) that

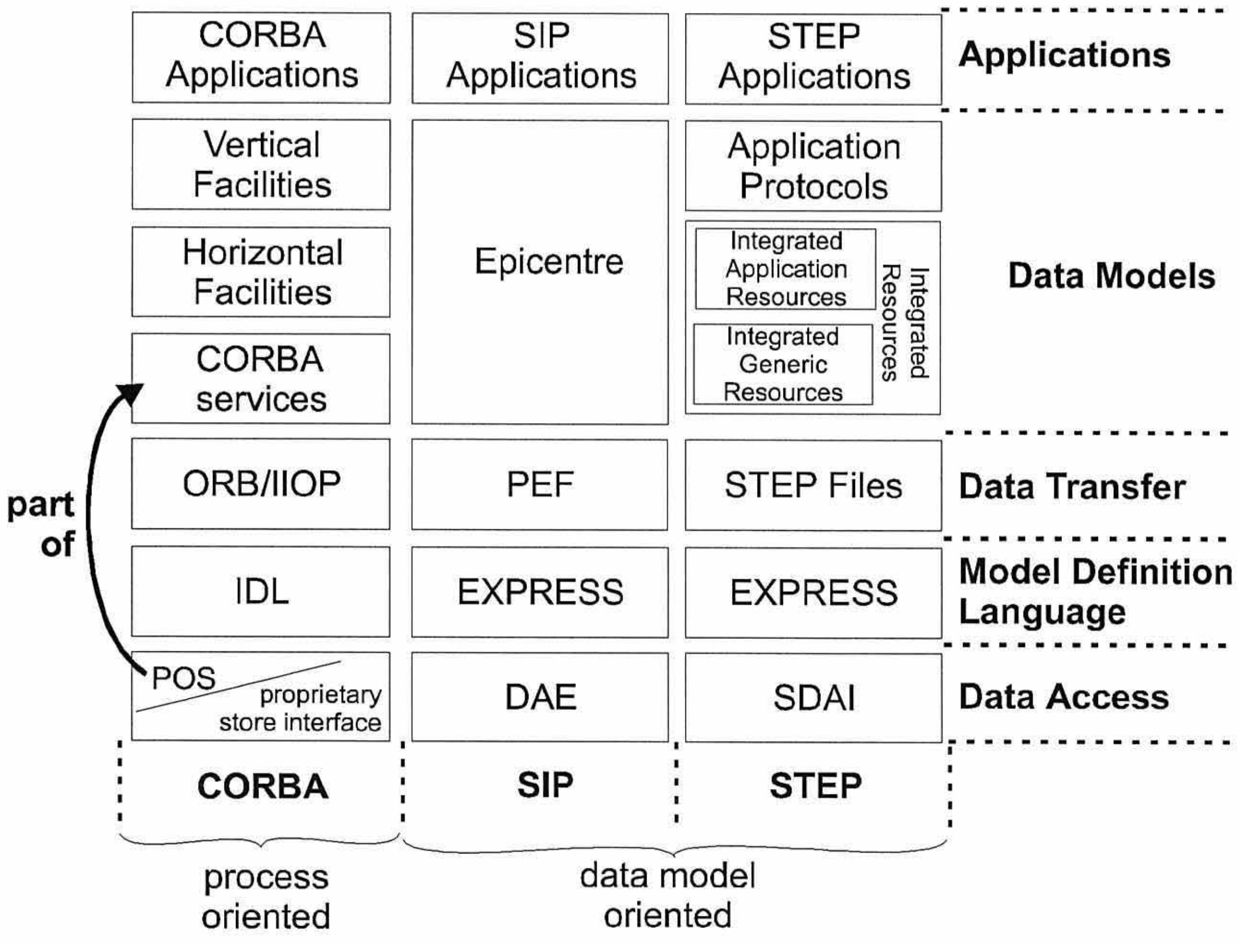


Figure 2.21: *Matrix of the components of each standard with similar functionality*

system parts(i.e. CORBA objects) can directly send messages to each other to carry out operations(Figure 2.24). The inter-operability of system parts is modelled using the Interface Definition Language(IDL). An interface definition specifies the operations that a system part implementing that interface can carry out.

CORBA also incorporates a suite of common service specifications that provide much functionality for systems to operate in the distributed environment. CORBA encourages the reuse and extension of these services. CORBA itself has built upon these services with the common facilities specifications that provide application level functionality.

A particular significant CORBA service is the Object Transaction Service(OTS). OTS permits multiple distributed heterogeneous applications to work within a single transaction. For systems that must be reliable, transaction capability is essential. Therefore, the combination of CORBA and OTS will be a strong candidate for the basis of future enterprise systems.

CORBA offers a sophisticated dynamic run-time inter-operability system that is largely process oriented. On the other hand, STEP and SIP inter-operability is based on the sharing and exchange of statically defined data. These data oriented applications concentrate on the structure, semantics and management of the data, rather than how the application can dynamically inter-operate with others.

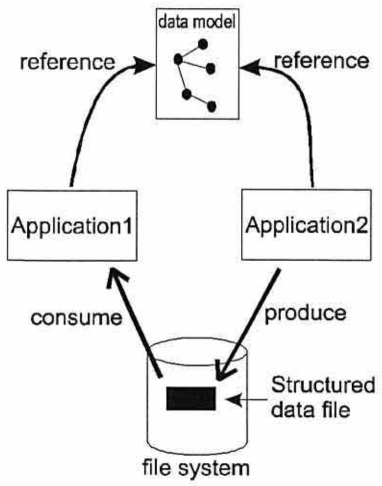


Figure 2.22: *Data exchange with a common data model*

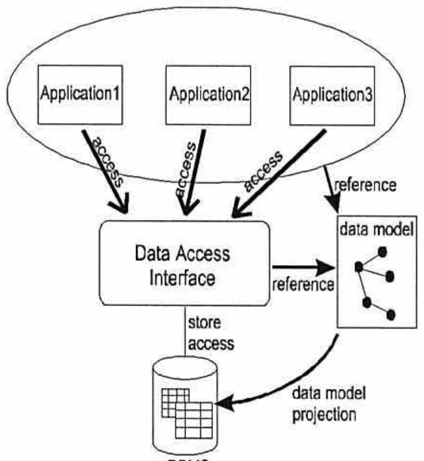


Figure 2.23: *Data sharing with a common data model*

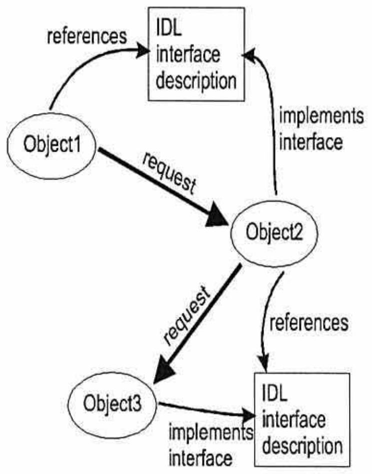


Figure 2.24: *Dyanmic inter-operability between system components*

CORBA is strong in dynamic interoperability, but weak in the management of complex persistent data. The reverse is true for STEP and SIP. They are strong in data management and weak in dynamic interoperability.

One of the great strengths of STEP and SIP is their use of EXPRESS as a data definition language. In CORBA, IDL could be used as a data definition language to define data models, but it does not have some of the powerful abilities of EXPRESS, such as the capture of relationships and constraints. These abilities are essential to create an unambiguous, fully specified data models.

Both STEP and SIP have a standard data access API. Applications built to the API specifications will be portable across implementation of the API. CORBA has no such equivalent suitable standard for data access. The Persistent Object Service(POS) was intended to meet this functionality, but failed due to design faults and being under specified. Consequently, CORBA developers are left to implement their own data access mechanisms to store data persistently using proprietary datastore interfaces e.g. file systems, DBMSs. This leads to more work for developers, less flexible systems, possibly more bugs and a decrease in application portability.

This section has discussed the strengths and weaknesses of the three standards. But the factor that each profits from is the ability to specify a model (whether data or interface model) that is independent of any implementation technology. Consequently, the logical models do not have to take into account the complexities and evolution of any specific implementation technology, thus the logical model will more accurately reflect the problem domain.

Chapter 3

Analysis of Current CORBA Persistent Data Access Solutions

3.1 Introduction

This chapter surveys current solutions that CORBA applications can utilise to access and manipulate persistent data. The persistent data is stored in some kind of datastore that is based on some specific storage technology e.g. files, relational databases, object-oriented databases and industry standard datastores. CORBA applications can access this data using some form of interface to the store.

Access to persistent data for CORBA applications can be divided into three categories. These are:-

- Proprietary storage technology interfaces.
- Standard IDL interfaces to storage technology.
- Persistent CORBA objects using database adaptors.

The chapter discusses each of these categories, including a description on how CORBA applications interact with the differing interface mechanisms and the benefits and

disadvantages of their use in conjunction with CORBA. The result of this investigation has influenced the proposed solutions presented in this thesis.
Included in this discussion are the following :-

- An overview of the OpenSpirit project. OpenSpirit is an excellent example of a three-tier architecture supporting components for the Exploration and Production industry. The data for the middle layer business objects have to be persistently stored. OpenSpirit's approach to persistent data access is examined.

- The Persistent Object Service(POS) was the OMGs answer to persistent data access. POS is extensively detailed as well as an in-depth analysis of its design and the major failures of its design.

- An overview of ISO 10303 Parts 22 and 26. Part 22 is the STEP project's Standard Data Access Interface, it specifies an interface to allow the manipulation of EXPRESS defined data. Part 26 is the binding of Part 22 to the OMG Interface Definition Language.

- Integration of CORBA database adaptors with object-oriented databases. This puts forward the argument that this kind of integration is the most elegant and ideal solution to access persistent data in CORBA, but for already existing data in "traditional" datastores this approach is not feasible.

3.2 Proprietary Storage Technology

The most common method of accessing persistent data for any application is through an interface of some proprietary storage technology. For example:-

- For a file system, data is stored in files and accessed using some programming language specific API e.g. 'C' standard input/output functions, 'C++' input/output streams.

- Accessing a relational database through a proprietary or standard Application Programming Interface(API) such as JDBC[SUN 97], ODBC[Orfali 96].

- Object-oriented databases(OODBs) such as Versant[OODB 1], Objectivity[OODB 2]. Also, one level stores such as ObjectStore[OODB 3], where memory and persistent storage are closely coupled, so that objects in memory are effectively persistent and all persistence operations are transparent to the client.

- An industry standard datastore and access interface e.g. the POSC Data Access & Exchange Facility(DAEF) supporting Epicentre and the Data Access & Exchange(DAE) API.

3.2.1 Problems with Proprietary Storage Technologies

To develop an application that requires data to be persistently stored, a developer will create a logical model of the data. This logical model can then be mapped to an implementation model that the selected storage technology can support. However, transforming the model to fit the storage technology creates an impedance mismatch between the application logical view of the data and its stored format.

The impedance mismatch can be defined as the amount of work required converting data between two differing models where these models semantically represent the same data. There are degrees of impedance mismatch that is dependent on how dissimilar models are. For example, mapping an object model to a relational database the impedance mismatch is high. The relational model does not have concepts such as inheritance, object identifiers, complex data types and direct object manipulation. There is a low impedance mismatch between an object model and object-oriented databases(OODB), as OODBs support all these concepts, thus the object model can be directly mapped to the OODB, requiring very little if any conversion of data.

A low impedance mismatch is an ideal situation to have between logical and storage implementation models. This is due to the great amount of code and processing that will be needed to transform data from one model to another when there is a high impedance mismatch. The need for additional code and processing to cross a high impedance mismatch leads to several additional problems:-

- Extra effort needed to create code, therefore extra cost.
- More possibility of bugs.
- Small changes in the application logical model will lead to large changes in conversion code.
- Advances in storage technology can incur bugs.
- Integrity of data is more difficult to maintain.
- Performance is decreased due to conversion operations and maintaining data integrity.
- Conversion code is generally not reusable and has to be thrown away should the storage technology change.

The result is very fragile code that is sensitive to changes in either model. Any changes will require maintenance in the conversion code that could introduce bugs that will need fixing and further testing performed.

The design of the mapping to the implementation model should be carefully carried out with the possibility of changes and extensions to the logical model taken in consideration. Otherwise, changes to the logical model will result in the implementation model becoming extremely complex and performance inefficient[Ambler 99].

Mapping between logical and implementation storage model is a major concern for developers. If not performed correctly then this can result in major delays in product release and vast expense in software maintenance.

Also of concern is the close cohesion the application code has with the proprietary storage technology. This mainly effects the portability and reusability of the code as the application can be locked into the specific vendor's storage technology product. To port the application to another vendors product requires a major rewrite. Another consequence of this product lock is that the developers choice of platform, operating system and programming language is limited to that for which the storage product is available. Therefore, reusability and portability of the application is poor.

For standard data access interfaces such as JDBC[SUN 97], ODBC[Orfali 94], DAE[POSC 95b] and standard file I/O functions product lock is not so much of a problem given that a developer abstains from any proprietary features of the specific datastore implementation. But there is still the impedance mismatch caused by mapping to consider.

CORBA applications will use such data access interfaces either to provide a form of persistence to CORBA objects or simply carry out some processing on persistent data and make the results available to the CORBA distributed environment. As the CORBA application is directly accessing the datastore through its proprietary interface, the application therefore inherits the problems of data mapping and vendor product lock as previously mentioned.

This close coupling[Somvil 89] of CORBA application/object and proprietary datastore interface is contrary to the nature of component architectures which is the very essence of CORBA. Component architectures entail the encapsulation[Rumb 91] of program complexity within an interface. Encapsulation permits maximum reusability, portability and a low coupling so that changes in one component have little or no effect on components that are inter-operating with the component through its interface. The linkage between CORBA application/object and datastore does not have these benefits and is therefore detrimental to the quality of the software.

3.2.2 OpenSpirit and its Data Access Approach

The OpenSpirit[Godfrey 97] project is an alliance of Exploration and Production(E&P) companies that are designing and developing a common framework of components to enable integration and data sharing of systems within the E&P industry. The project is initially concentrating on components representing subsurface features e.g. wells, wellbores and associated measurement of their features such as log traces and seismic surveys. These components are realised in the form of IDL interface definitions, thus all communication between components/CORBA objects is through an ORB.

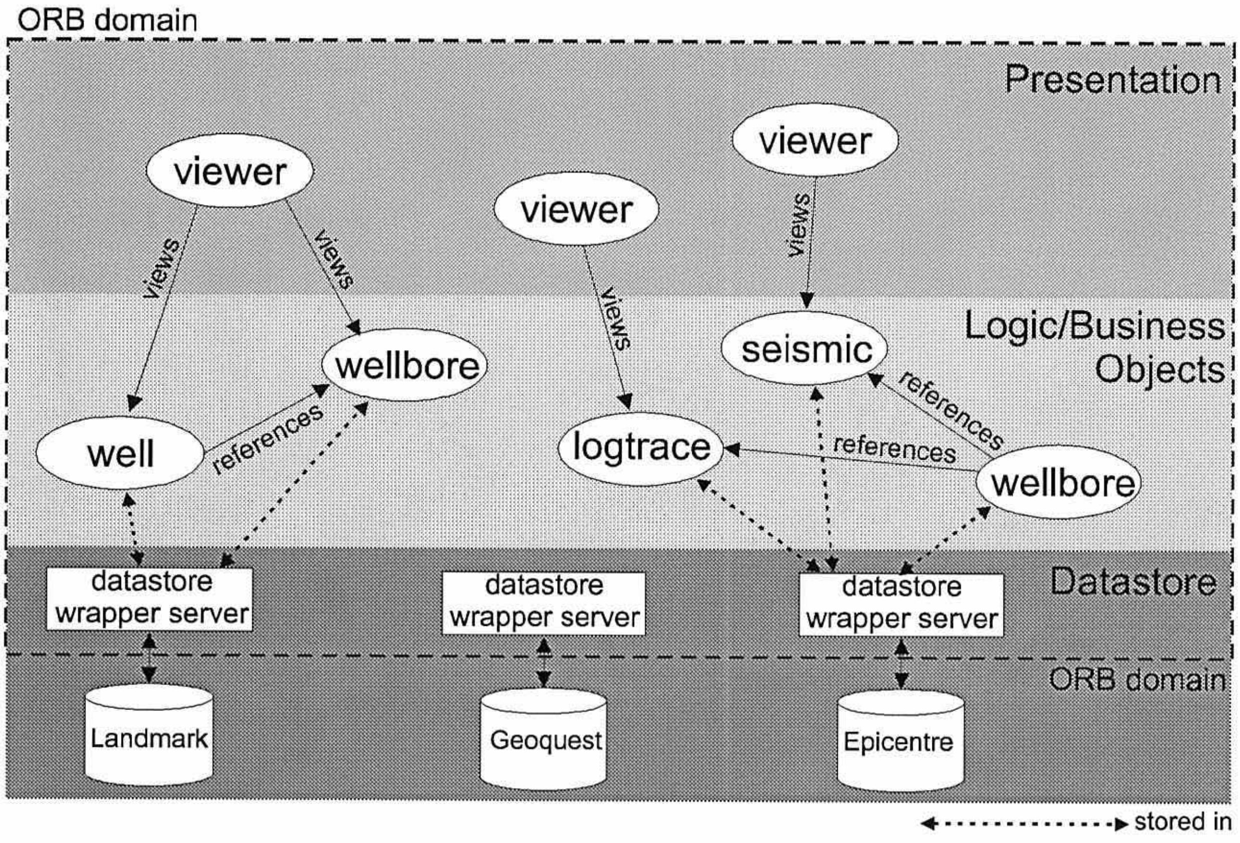


Figure 3.1: *The three tiers of the OpenSpirit architecture*

The architecture of the OpenSpirit component framework follows a three tier architecture[Shan 98](Figure 3.1). The middle logic layer comprises of CORBA objects/components representing common subsurface features. The presentation layer contains viewers that provide a view to the user of these middle layer components. The datastore layer comprises of IDL wrappers to popular E&P datastores. Each component relies on an IDL wrapper to an E&P datastore to preserve its state.

The datastore layer consumes a large portion of the development effort of the implementation of the OpenSpirit framework. This is due to the effort needed to map each component from its implementation model to a form where it can be served to the business object across an IDL interface.

Each type of OpenSpirit E&P(OSEP) object has an equivalent IDL interface that acts as a datastore wrapper. An implementation of this IDL wrapper permits an OSEP object to request and set data that constitutes its persistent state. Several layers of software are required to implement the IDL datastore wrapper shown in figure 3.2. The upper most layers provide the connectivity between the OSEP object and its datastore wrapper interface using the ORB. The specific implementation of the datastore wrapper methods are written in Java and utilise a Java ORB for distributed communication. OpenSpirit has taken this approach for the following practical reasons:-

- Portability of code due to Java's write once, run anywhere ability.
- Only one single ORB run-time in use.
- Object implementation code can be reused for each E&P datastore in use with the OpenSpirit project.
- The OpenSpirit project makes use of a set of services called the Business Object Facility[Prism 99] by PrismTech[PrismTech]. These services are written in Java, hence they can be used locally rather than remotely as would be the case if the wrapper was written in another language.
- Easier distribution of OpenSpirit software.

The drawback of this approach is that the access interfaces to the supported E&P datastores are based in C or C++, therefore a cross-language integration mechanism is

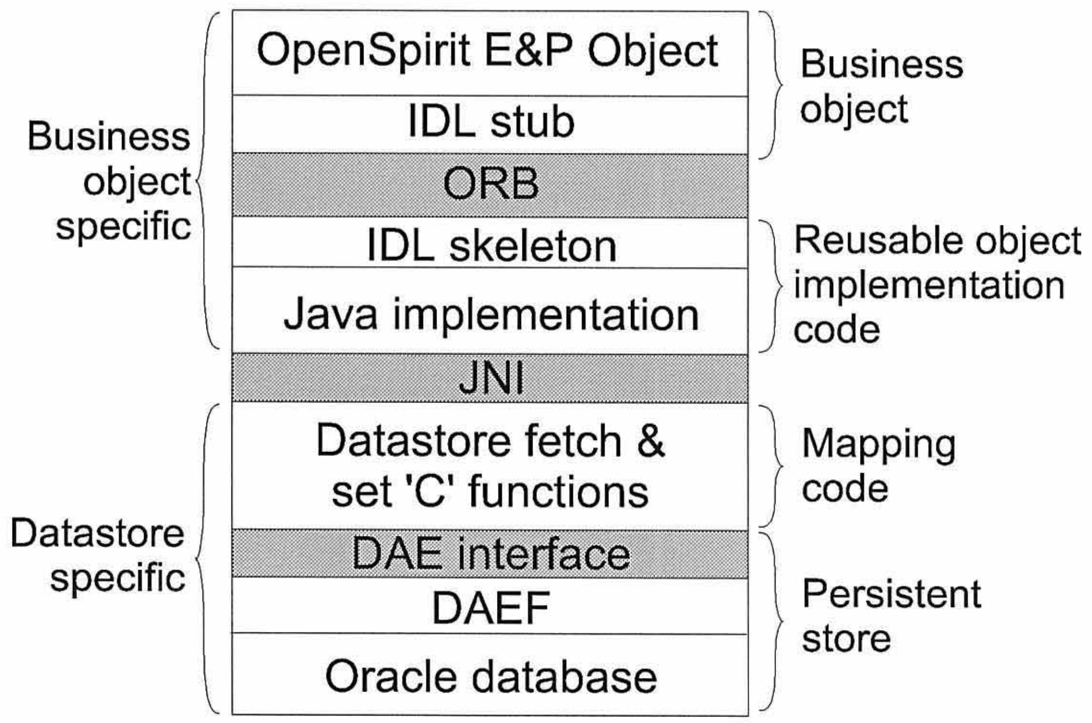


Figure 3.2: *Layers of software implementing an IDL datastore wrapper*

necessary. Java provides integration with C based languages using the Java Native Interface(JNI)[SUN 97b]. JNI allows Java programs to call native C functions including the passing and returning data. JNI also allows native functions to share memory management of the Java virtual machine.

The mapping code layer provide 'get' and 'set' C language functions. These contain the intelligence to convert data between the datastore model and the logical model of upper layers, thus spanning the impedance mismatch.

The mapping between objects in the upper layers and data entities in the datastore will not necessarily be a one-to-one mapping. In considering the mapping of OSEP objects to the Epicentre data model, each OSEP object will be mapped to many Epicentre entities. This is due to the fact that OSEP objects are very coarsely grained and Epicentre entities are very finely grained, therefore attributes of OSEP objects map to many individual entities that are intricately interconnected by relationships. To retrieve (or set) OSEP object state data, these relationships have to be navigated and the data contained in the Epicentre entities have to be pulled out and transformed to a format for passing across the JNI.

The implementation of the OpenSpirit E&P mapping to Epicentre was personally carried out by the author of this thesis. This work was completed within the employment of PrismTech[PrismTech]. PrismTech is the development and marketing partner of the OpenSpirit alliance and is part sponsor of this project.

An area that PrismTech specialises in is providing data management and integration tools for the E&P industry including LightSIP - an implementation of the DAE supporting Epicentre, and conversion tools to convert common data formats to an Epicentre format. PrismTech is working towards a complete solution for E&P data management in the form of a data warehouse. This will involve the conversion of data between many different industry formats and models. To meet this challenge PrismTech is developing the Mapping Manager Architecture[Prism 98].

The Mapping Manager Architecture(MMA) is a methodology for moving and transforming data between different data models. The MMA specifies a three-stage process to convert a data between two data models. The first stage is a formal definition of the map between the data models, second is the automatic generation of executable code to carry out the conversion, and the final stage is the execution of this code to transfer data between the two models. The pivotal element of this architecture is the formalization of the mapping between data models. This is accomplished by specifying the mapping in PrismTech`s Expressive language[Prism 98b].

The Expressive language allows the formalization of the mapping from one model to another. It achieves this by permitting the definition of how attributes of a data entity or entities in a source model are assigned to attributes of a data entity or entities in a target model. The language can also be used to define control over instance creation of entities in the target model. Navigation of relationships between entities is also permitted, thus attributes of many different entities can be extracted to fulfill the required data for attributes of a single entity of the target model. These are only the core features of Expressive, as it contains many other features to help formalise the complex task of specifying how one data model maps to another.

The powerful features that Expressive provides made it an ideal facility to specify a formal mapping between OpenSpirit E&P objects and the Epicentre data model. The Expressive mapping for OSEP objects to Epicentre was designed by Steve Trythall of PrismTech E&P data management division. To design such a mapping requires a comprehensive knowledge of both models including the structure and semantics of the models. This is especially true in the case of Epicentre where there can be multiple sets of entities that OSEP data can be mapped to, but the design semantics of Epicentre dictate only one set of these entities.

Stage two of the Mapping Manager Architecture is the automatic generation of executable code from the Expressive mapping to transfer data between the two models. To achieve this, two pieces of conversion code need to be generated. The first piece translates the data in the source model to a standard model format. The second piece translates the standard model format data to the destination model. To develop code generator tools to produce such translation code requires intimate knowledge of each respective data model and the access interfaces at each end of the translation. For Epicentre, this would be extremely complex for the following reasons :-

- Epicentre has many extended data types that have a complex structure to hold data such as spatial, quantity and geometric (Mapping these data types is complicated as well as saving them in the DAE).
- Each Epicentre entity has a set of attributes that makes up the entity's natural identifier. The combination of these attributes is unique within the domain of all instances of that entity type. This uniqueness has to be maintained for successful translation.
- Many Epicentre entities have mandatory attributes. These attributes are not allowed to have a null value. Sometimes the need for this data is not mandatory in other models, therefore will not exist. In this situation, mapping code would have to create arbitrary values for these attributes.
- Many Epicentre entities have constraints placed on their values by rules declared in the EXPRESS procedural language, that have to be complied with.
- The DAE interface is very complex to use, especially the frames facility that manages scientific and spatial data.

Due to these reasons, a massive amount of development time and resources would be required to create an Expressive automatic translator from a standard data model format to Epicentre. Therefore, the OpenSpirit to Epicentre/DAE mapping was manually coded and a direct translation strategy was selected with no intermediary standard data model format.

Development of the get and set functions to map each OSEP object to Epicentre was an awkward and time-consuming process. The first stage of this development was the manual creation of test data within the POSC datastore. The test data required the creation of entities involved the mapping, as well as the creation of entities that are the mandatory attribute values of these mapped entities. Given this test data, each get function has the following stages :-

1. Start a transaction.
2. Use a query to search for a target entity that is identified by key information.
3. The target entity will typically have references to other entities that hold data required for the mapping. Traverse the relationships from the target entity specified in the Expressive mapping to gain references to the required entities.
4. Copy and convert the data from the target entities to the JNI data structures. This stage is not a simple matter owing to the fact the data is made up scientific values or spatial data and is contained in DAE frames. Reverse this stage for set functions.
5. Finish the transaction.

The development of the get functions was used as an evaluation of the correctness and consistency of the Expressive mapping to Epicentre. The consequence of this was slight changes to the mapping design. The next stage of development was the creation of the set functions to write data from OSEP objects to their persistent Epicentre form.

The get and set functions carrying data between OSEP objects and Epicentre entities provide the principal functionality to cross the impedance mismatch between the Epicentre model and the OpenSpirit model. The functions carry out the greatest amount of work to provide persistence to the OpenSpirit components and take up the majority of development time for the datastore layer, especially in considering that each component has to be mapped to each datastore type supported by the OpenSpirit project.

The OpenSpirit project and its approach to providing persistence to components/CORBA objects is typical of current solutions concerning the provision of access to persistent data from CORBA applications. This solution involves the wrapping of datastores in an IDL interface and providing mapping code as the implementation of the interface to translate the implementation datastore model to the application logical model. Although the middle tier components are disjoint from their mapping code via the IDL datastore wrappers, the wrapper implementations still incur the problems with proprietary datastore interfaces described in 3.1.1.

OpenSpirit acquires some additional problems due to the design of its persistence architecture. The use of the Java Native Interface(JNI) to bridge ORB specific Java implementation code and mapping code introduces great complexity into the persistence architecture. The JNI approach to language integration is not as clean, automated and defined as that of CORBA integration using IDL interfaces. The native side of the JNI has to manually convert data to a format that is acceptable to the Java virtual machine. Use of the JNI adds an additional impedance mismatch, thus making the code more bug prone, harder to maintain and sensitive to model changes. To preclude the use of JNI, a C/C++ ORB could be used, but this would counter the initial reasons for choosing a Java based ORB.

The high level view of the OpenSpirit persistence architecture is a rather simplistic one. This view is of components providing datastore wrappers with identifier information to locate their state data. Once located, the wrapper fetches the data and returns it to the component. The significant features missing from this view is that of concurrency and transaction capability that are essential for any reliable system. To have such a capability, the clients of the datastore wrapper must allow some sort of session to be set-up that relates the client with its activates in the datastore. The session will give access to the locking and transaction functions of the datastore as well as keeping track of locks and transactions that the client is responsible for.

Currently, OpenSpirit does not control concurrent access to datastores. This situation is fine for simply distributing data in a standardised form for read-only access, but for a fully functional read-write system, concurrency and transaction control is vital.

As mentioned, the OSEP objects are fairly coarsely grained. Each OSEP object has a counterpart datastore wrapper interface and implementation to provide it with access to its persistent state. As there is a low number of OSEP objects, there are only a few datastore wrapper interfaces. If the granularity of OSEP objects was to increase or if we might wish to provide access to each Epicentre entity from CORBA, then to use this persistence model each object would have a datastore wrapper interface and manually coded conversion code. For a data model the size and complexity of Epicentre, this approach would be unfeasible.

This is due to the amount of developer time and effort required to create bug free code to convert each Epicentre entity to a CORBA object form.

Another problem with large, finely grained models is how references between entities/objects are handled. In OpenSpirit, references are handled by an OSEP object passing simple identifier information to the datastore wrapper. For example, a `well` entity has a string identifier that acts as a unique name for the well. An OSEP object will pass this identifier to the datastore wrapper to identify its persistent data. For more complex models referencing of this type is not always possible for the following reasons:-

- An entity might not have a unique identifier, but is identified by navigating a relationship path from some other identified entity.
- An entity might be part of one-to-many relationship. In this case the single entity would keep attribute identifiers to each entity it is referring to. If there is a large number of entities in the many relationship then the single entity would be very large with all the identifiers it has to keep.
- Referential integrity is harder to maintain as identifier information is spread amongst many entities.
- Performance can be negatively impacted using unique attribute identifiers as a search is necessary to find the referenced entity.

In summary, OpenSpirit is the first of its kind in the Exploration and Production industry. The project is providing a three tier architecture with components in the middle layer representing common business entities that are independent of any E&P datastore, thus are also providing a form of integration between the popular E&P datastores. Each component interacts with other components using their IDL interfaces and the ORB, hence components of the OpenSpirit framework are truly distributed and open allowing reusability with ease. However, the OpenSpirit component framework is a rather simplistic model. This has the effect of making mapping to datastores easier. But even at this level of sophistication crossing the impedance mismatch between logical model and implementation model is a hard, time intensive and error prone task. For models such as Epicentre that are very finely grained, the mapping is even more difficult owing to the numerous Epicentre entities involved in a single mapping as well as the many constraints that are put on Epicentre entities. An ideal situation would be to automate the mapping task. PrismTech's Mapping Manager Architecture with its Expressive mapping language seems to be a natural solution.

3.3 Standard Interface Definition Language Specifications to provide Data Access to Persistent Storage

This section surveys data access interfaces and services that are specified in IDL and are international standards. The first standard specification is the OMG Persistent Object Service(POS). POS is the OMG's solution to providing persistent data access in the form of a set of services that provide persistence to the state data of CORBA objects. The second standard is a specification from the STEP project that provides a standardised data access interface to EXPRESS defined data. This interface has an IDL binding allowing data to be manipulated from CORBA applications.

3.3.1 The OMG Persistent Object Service

The OMG Persistent Object Service(POS)[OMG COSS][Session 96] is one of the OMG CORBA service specifications. POS was the OMG's solution to providing persistence to CORBA objects, hence should be the main mechanism to provide access to persistent data. As with other CORBA service specifications, the POS specification provides a set of IDL interfaces and an explanation of how components implementing these interfaces work together to meet their aim i.e. providing a persistence service.

A key design principal of POS is that POS should support both traditional datastores e.g. files, relational databases as well as newer data storage mechanisms i.e. object-oriented databases to store persistent data. The reason behind this design decision is that the vast majority of corporate data is stored in these traditional datastores. Figure 3.3 is taken from [Session 96] and shows an estimate of the amount of data stored in files, traditional databases and object-oriented databases. The estimate of the amount of data stored in object-oriented databases is minuscule compared to files and traditional databases.

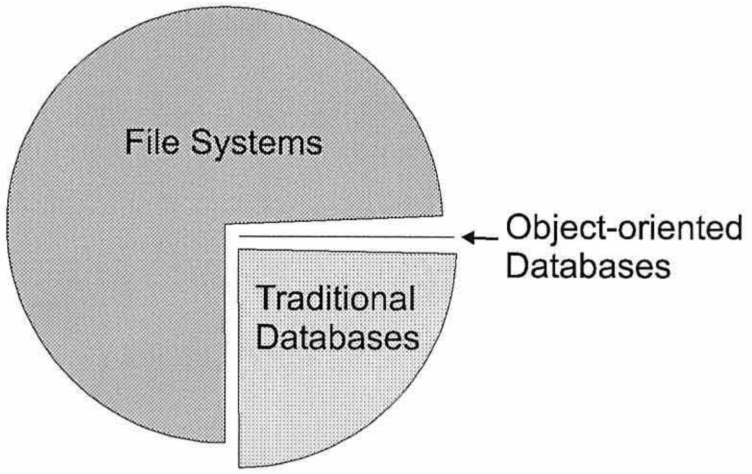


Figure 3.3: *Estimate of amount of data stored in different datastore types*

Therefore, to simply support object-oriented databases, which is CORBA's most similar storage model would be a serious drawback to the future growth of CORBA.

However, the design of POS to support persistence in a broad range of datastores was the primary reason why it failed and has been withdrawn by OMG as a standard. This is due to the fact that developers of POS implementations found POS un-implementable owing to major design flaws and being greatly under specified. This section describes the design of POS and then continues with an examination of the reasons why POS has failed and what can we learn from it.

3.3.1.1 The Design of the Persistent Object Service

The Persistent Object Service(POS) regards a persistent object's state as having two parts. These are a dynamic state and a persistent state (Figure 3.4). The dynamic state acts as the object's state data in memory and only exists for the lifetime of the object. The persistent state is state data stored in a persistent datastore and is copied out to memory to create the dynamic state, thus survives multiple creations of the dynamic state. The IDL interfaces of POS (Appendix B) are used to identify, create and delete persistent states as well as to move the state data between dynamic and persistent states.

As mentioned, the persistent state can be stored in any type of datastore. The only constraint on the type of datastore and the implementation model of the datastore is that the persistent state must be able to be uniquely identifiable for the purpose of finding the persistent state and to read or write it.

To move state data between persistent and dynamic states, POS utilises a protocol. A protocol is a structured way to transport the state data from the datastore to the persistent object using requests. The POS specification defines no standard protocol, but gives an example of three possible protocols.

Excluding interfaces concerned with protocols, there are four fundamental POS interfaces shown in figure 3.4. These are the Persistent Identifier(PID), Persistent Object(PO), Persistent Object Manager(POM) and the Persistent DataService(PDS).

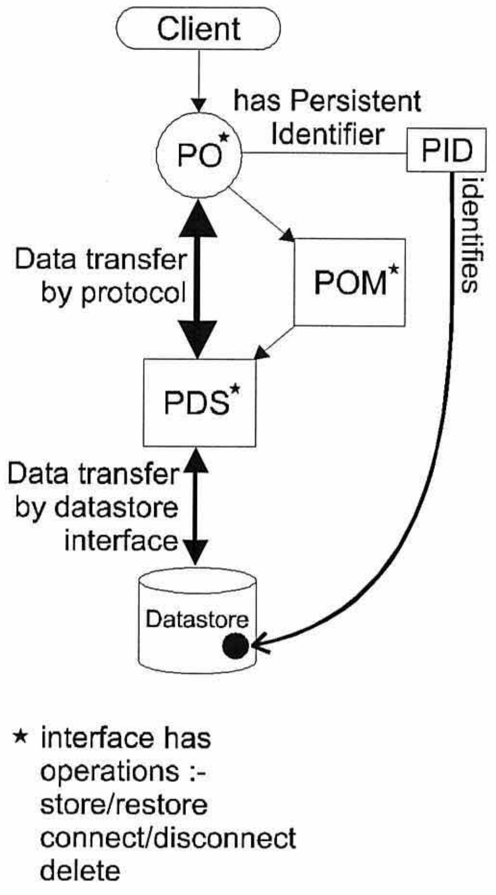

Client
has Persistent Identifier
PO*
PID
identifies
Data transfer by protocol
POM*
PDS*
Data transfer by datastore interface
Datastore
★ interface has operations :-
store/restore
connect/disconnect
delete

Figure 3.4: *Structure of the fundamental interfaces of the Persistent Object Service*

The Persistent Identifier(PID) interface provides a base interface to allow the identification of the persistent state in the datastore. This interface is intended to be extended for each datastore type. The extensions of the interfaces will provide additional attribute information to identify persistent states for specific datastore types. For example, a file path name could be added to identify a file or some unique object identifier identifying a persistent object in an object-oriented database.

The PO, POM and PDS interfaces provide the same operations in their interfaces. These operations are connect, disconnect, store, restore and delete. Each operation has a PID reference as a parameter. Additionally, both POM and PDS have a reference to a PO object as an extra parameter. The POS specification stipulates that the components implementing these interfaces act in a chain. That is, a request of one of the operations of the PO interface, results in the PO component invoking the same operation on the POM and the

POM carries on the chain by invoking the same operation on the PDS component. However, each component has a different duty to perform on receipt of a request.

The Persistent Object(PO) interface is implemented by the actual object whose persistence is being handled by the Persistent Object Service(POS). This interface lets a client control the persistence of the object. In addition to the operations just indicated, the PO interface has an attribute that references a PID object. It is the task of the client set location information contained in the PID to identify the persistent state of the object. Once the PID is set, the other operations of the PO interface can be used to control the persistence of the PO. These operations have the following semantics upon the object's persistence :-

- connect/disconnect : these operations create and break a connection between the dynamic state and the persistent state, so that any updates in one are reflected in the other.
- store/restore : these operations allow the explicit loading and saving of the dynamic state to/from the persistent state.
- delete : the removal of the persistent state from the datastore.

The Persistent Object will forward an operation invoked upon it to the Persistent Object Manager(POM). It is the duty of the POM to locate the Persistent DataService(PDS) that is a wrapper for the datastore holding the POs persistent state and forward the request to it. As mentioned, the PO and PDS will use a common protocol to transfer data between them. It is also the task of the POM to choose a PDS that can "speak" the same protocol as the PO.

The Persistent DataService(PDS) is responsible for getting data in and out of its datastore. It is also responsible for getting this data in and out of the Persistent Object using a specific protocol. It is the PDS that carries out the processing to convert the implementation model of data to the logical model, consequently crossing the impedance mismatch. Once in its logical form the data can be pushed or pulled to/from the Persistent Object using a protocol.

To further reinforce and clarify how POS works, the following scenario explains how a Persistent Object retrieves its persistent state using the "restore" operations.

1. The client creates the Persistent Object(PO) using an object factory.
2. The client sets the Persistent Identifier(PID) attribute of the PO to indicate the location of its persistent state.
3. The client invokes the restore operation of the PO interface.
4. The Persistent Object invokes the restore on its Persistent Object Manager(POM).
5. The Persistent Object Manager somehow decides on the Persistent DataService(PDS) containing the identified persistent state and communicates with the Persistent Object using a specific protocol.
6. The POM invokes the restore operation on the Persistent DataService.
7. The PDS component retrieves the identified data from its datastore.
8. The PDS transforms the data to a form that allows it to be passed to the PO using the supported protocol.
9. The PDS transfers the data to the PO using its protocol.

The POS specification proposes three possible protocols for moving data between the PDS and the PO. These are the Direct Access, ODMG-93 and the Dynamic Data Object protocols.

The Direct Access protocol allows a Persistent Object to directly read or write using attribute get and set requests to a CORBA object. This CORBA object is called a Data Object and is instantiated by the PDS. The Data Object gives access to the PO's persistent state via attributes declared in its interface.

The ODMG-93 protocol is similar to the Direct Access protocol, but is based on the ODMG's object database standard. The protocol uses the ODMG's object definition and manipulation languages instead of IDL and attribute get/set requests.

The Dynamic Data Object protocol is a variant of the Direct Access protocol. This protocol uses a Dynamic Data Object instead of a static IDL interface to access data. Thus, the protocol extracts data by examining attribute descriptions and extracting the data from IDL 'Any' types.

The final part of POS provides standard IDL interfaces between the PDS and record-oriented databases e.g. relational, hierarchical and VSAM file systems. This interface is based on the X/Open Call Level Interface[Orfali 94] standard and is called the Datastore_CLI. The Datastore_CLI interfaces provide operations to set-up a connection with a datastore and allow creation, deletion and manipulation of record data using cursors.

To summarise, POS provides a framework of interfaces to identify, load and save the persistent data of CORBA objects. Movement of data between the Persistent Object and the Persistent DataService is provided by one of the suggested protocols or a proprietary protocol. The main strength of this framework is the fact that it does not discount the use of any type of datastore in favour of a datastore with a closely related model to that of CORBA i.e. object-oriented databases. However, the POS framework is only an abstract model of how persistence can be achieved. POS cannot be directly implemented.

3.3.1.2 Problems with the Persistent Object Service

The design of the Persistent Object Service(POS) seems like an excellent solution to provide persistence to CORBA objects in a broad range of datastores. Nevertheless, to actually try to design an implementation of POS reveals many problems. These problems become more apparent the further the POS specification is examined.

Some problems with POS are mainly due to its major under-specification. This under-specification allows POS the flexibility to support a range of datastores. To compensate for this under-specification a developer of POS would have to design their own solutions to make POS implementable. Consequently, the POS implementation would become proprietary to that developer, thus would not have the benefits of portability and reusability that are the principal aims of common object services.

Other problems with the POS specification are major design flaws innate its architecture and ambiguity in the description of some parts of the specification. The compound effect of all these problems led to no commercial implementations of POS being developed, resulting in the OMG withdrawing the specification as a standard.

The following section gives a brief account of some of the more significant problems of the Persistent Object Service.

• Transaction and Concurrency	The specification makes no attempt to describe how POS interacts with the Object Transaction and Concurrency services. These services are essential to any reliable system and are especially significant in any system dealing with persistence. In POS, it is not known how a Persistent Object will lock its persistent state to prevent other incarnations of itself manipulating the data. Also, how will POS components react to a transaction commit or abort?
• Object Referencing	POS only offers a weak form of object referencing either through a CORBA object reference in its string form or by using Persistent Identifiers(PIDs) encoded into object attributes.
• Relationship Service	POS suggests that the Relationship service[OMG COSS] be used to handle relationships between CORBA objects and make use of the Relationship service's TraversalCriteria object as a method of storing a graph of objects as a single unit i.e. transitive closure of dependencies. How this integration of services is achieved is not explained. An investigation into this integration of services was carried out in [Klein 95][Klein 96][Klein 96b]. The investigation found many problems in this integration, including making the relationship object themselves persistent, performance exceedingly degraded due to the dereferencing needed to traverse a relationship, and how is a TraversalCriteria specified.
• Underspecified functionality of the Persistent Object Manager	For the Persistent Object Manager(POM) to decide on the correct PDS to serve the object's persistent state, the POM needs to know two types of information :- the protocol the PO supports and the protocol of each PDS known to the POM. POS does not indicate how this information can be found, but [Sessions 96] suggests several possible solutions such as :- examining the interface of the PO to see if it supports a protocol interface, POM offering operations to register protocol-PDS associations and examining the interface repository.

• Controversy in the semantics of the connect/disconnect operations	POS explains the semantics of these operations as "The persistent state may be updated as operations are performed on the object". Does the word 'may' mean updates will be forwarded to the persistent state or might be forwarded? The inclusion of these operations is suggested by [Sessions 96] for the purpose of supporting single level stores such as ODI's ObjectStore[OODB 3], where memory and storage are tightly coupled offering a virtual persistent memory. [Sessions 96] believes these operations are not implementable even with a single-level store due the 'connect' operation which is executed on an already instantiated object, however single-level stores provides a persistent connection when the object is instantiated. The author of [Sessions 96] is one of the main architects of POS and recommends connect/disconnect not to be used due to its vague semantics and an efficient implementation of these operations is difficult, if not impossible.
• Persistent Identifiers (PIDs) for non-simplistic identifiable data	POS relies on data being uniquely identifiable. Information for this unique identification is stored in the Persistent Identifier(PID). POS gives some examples of implementations of PIDs such as a path name for files, or an object identifier for objects within an object-oriented database. For non-uniquely identifiable data, for example tuples in a relational table, POS recommends a string containing a SQL statement. This is fine for very simplistic data models, but for data models with sophisticated relationships like multiple 1-to-many relationships, the PID will become increasingly more complex to use.
• Blocking problem	A severe problem inherent in the design of the POS architecture is the problem whereby the Persistent Object(PO) blocks due it performing a request. When a 'restore' request from a PO reaches a Persistent DataService(PDS) component, the PDS retrieves the data from its store and then is required to transfer the data to the PO using a protocol. But the PO process is blocked as it is executing a request, therefore a deadlock state is created. To solve this problem the PO process could be multi-threaded, but this would entail an immense amount of additional complexity for every application using POS.
• Inefficient protocols	The suggested protocols in POS are very inefficient, as each attribute of an object will require one or more requests to transfer the data to the Persistent Object.
• Inconsistent design of the Direct Access	The Direct Access protocol goes against the design principals at the heart of POS. As the Direct Access protocol extends the PDS

protocol	interface to provide operations to more than just transfer data e.g. getting and setting root objects.
• Persistent Object must drive protocol	The suggested protocols depend on the Persistent Object making requests to push and pull the data to/from the PDS. Therefore, it is necessary for the PO to hold the code to perform this and to convert the data to the PO's own attributes. This results in the object containing a large amount of code that is sensitive to logical and protocol models. Also, the responsibility for writing this code is given to the developer of the PO, thus extra effort is needed by the developer and could introduce errors. A more ideal situation would be to let the persistence system automatically take care of all data transfer and conversion, leaving the developer to concentrate on the application logic.
• No standard protocol	As POS does not specify a single standard protocol to transport data, developers will implement their own proprietary solutions, consequently weakening the standard.
• No exceptions	None of the POS operations are defined as being permitted to raise exceptions. Therefore, if an operation was to fail, the application would have no way of knowing about it. This is a major oversight in the design of the specification.

The above list shows only the major faults within the POS specification. There other small peculiarities within it, like the reasoning behind certain operations, their parameters and their intended behaviour.

As mentioned, developers would have to design their own solutions to solve these problems. Accordingly, the POS implementation would simply become another proprietary interface to datastores with the POS framework lending little value to the implementation.

After examining the POS specification in great detail[Sessions 96], the specification seems be very ill thought out and poorly designed. Some of the problems of POS come from it being designed by an amalgamation of two groups of designers. Each group had its own view of how persistence should achieved for CORBA components. One group's view was that object-oriented databases should integrate into CORBA to provide persistence, while the other group believed that persistence could be supported by any type of storage facility.

The greatest problem of POS is that it does not seem to have been verified that it works before becoming a standard. Implementing it first would have shown many errors and many poorly designed parts that would have brought the designers to the conclusion that it does not work.

3.3.2 The STEP IDL Standard Data Access Interface

The STEP[STEP 94](section 2.5) project has specified a set of standards to allow the definition, exchange and sharing of product data. The fundamental element of STEP is the ability to unambiguously define the structure of data using the powerful concepts embodied in the EXPRESS[EXPRESS 92] information modelling language.

To access and manipulate instance data of EXPRESS models that reside in datastores, STEP specifies the Standard Data Access Interface(SDAI) specification(ISO 10303-22)[STEP SDAI]. The SDAI is a language independent interface specification for data access. The SDAI has similar functionality to that of the POSC Data Access and Exchange interface, but is not exclusive to a single data model and is being implemented in a range of programming languages.

The SDAI details functions to manage the following :-

- Instance creation and deletion.
- Get and set attributes.
- Aggregates.
- Forward and inverse references.
- Details of entity hierarchy e.g. is kind of.
- Sessions including error reporting.
- Transactions and Concurrency.
- Validate rules.
- Entity instance searches.
- Meta information via dictionary data.

SDAI offers two different approaches to manipulating data from an application, early and late binding. The early binding approach is performed at compile time and entails the generation of static language constructs in the implementation language of the SDAI e.g. C++ classes. These language constructs are used to represent data being manipulated, thus acting as proxies for the stored data. The late binding approach allows dynamic manipulation of data at run-time. The data is handled by referring to dictionary data to find the structure of data. The early binding approach is simpler than late binding, but requires the extra stage of generation of language constructs. However, late binding is more generic in that it can handle different data models and is tolerant to changes and extensions in data models.

The mappings of the SDAI to programming languages are called the SDAI language bindings and are presented in additional STEP parts :- Part 23, the C++ language binding; Part 24[STEP SDAIb], the C language binding. A future Java language binding is in development. SDAI is also mapped to OMG's IDL(Part 26)[STEP SDAIc]. Which, although not a programming language, in turn has its own mapping to programming languages. This makes the SDAI available to any language that IDL maps to. SDAI also gains from the heterogeneous distributed nature of CORBA, therefore making a powerful marriage of technologies.

Using Part 26, IDL SDAI binding lets CORBA applications create and manipulate the complex data structures defined in EXPRESS. This permits CORBA applications to share data and inter-operate by understanding the semantics of the data, rather than simply passing data structures in the parameters of IDL interface operations. The IDL SDAI fills a gap that is missing from the CORBA services in manipulating complex persistent data.

The next question to ask is whether the IDL SDAI should be a CORBA persistent data access service. The reasoning against this proposal is described in the following :-

• SDAI is based on STEP technology	The concepts of SDAI are all based in the STEP paradigm including operation naming, error codes, data modeling language, consequently providing an extra paradigm for developers to come to terms with.
• Uses EXPRESS as a data modeling	EXPRESS is the data modeling language of the SDAI, thus only logical models described in EXPRESS can be manipulated. Data

language	modeling languages such as SQL, OQL and IDL are not considered.
• EXPRESS types only	The only data types permitted are those that are EXPRESS types. This is a problem with data models such as Epicentre that have many extended types.
• No integration with other CORBA services	The IDL SDAI does not integrate or reuse other CORBA services such as Object Transaction, Concurrency, Collection and Query services.
• Performance inefficient	Current implementations of IDL SDAI[Sauder 97][Amar] have only a thin client layer. All manipulation of data are performed using requests across the distributed environment to the datastore. For applications such as CAD/CAM that can have millions of small data objects, using this architecture is not viable. To solve this problem, caching is needed i.e. moving the data locally and manipulating it there.
• Implementations of the IDL SDAI have no standardised component structure	The SDAI is solely an interface specification, there are no details describing the components that make up the implementation such as integration with other CORBA services and how can data be cached.
• Does not take into account providing CORBA persistence	The SDAI is an interface for accessing persistent data, it does not indicate any way to use it to store CORBA object's persistent state, except by the manual manipulation of the interface by a CORBA object.

The SDAI and its IDL mapping is a fine means of accessing persistent data and sharing it among CORBA applications, but for a general CORBA service the SDAI does not fit well due to its foundation in STEP and non-composite design.

3.4 Persistent CORBA Objects using Database Adaptors

CORBA objects can be made persistent by being managed by a database adaptor. Database adaptors integrate databases into ORBs by extending the functionality of standard CORBA object adaptor(see section 2.3.3) to provide persistence for CORBA objects that are registered with the adaptor. The purpose of the database adaptor is to store the persistent state of the CORBA object within its supported database(Figure 3.5). The benefit of this

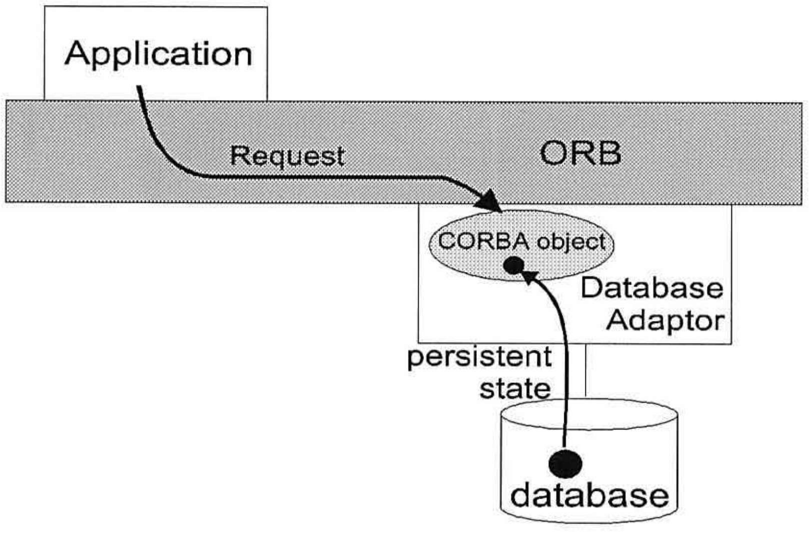


Figure 3.5: *Database adaptor storing CORBA objects persistent states in its database*

approach is that clients are unaware whether server objects are persistent or not, therefore do not incur the complexity of controlling the persistence of their server objects.

The database adaptor solution to persistence has already been proposed by the OMG[OMG 95] and the ODMG[ODMG 93] in their relevant specifications. Products providing this functionality have been very successful such as Iona's Orbix+ObjectStore Adaptor[OOSA 97] and Orbix+Versant Adaptor[OVA 97].

3.4.1 Complexities involved in Database Adaptors

There are more complexities[Reverbel 97][Amirb 97] to database adaptors than simply storing an object's persistent state, including :-

- Activation of objects
- Object references to persistent objects
- Mapping object states to the database
- Integrating IDL skeleton and persistent object implementation
- Mapping OTS transactions to database transactions

Persistent objects have to be activated to be able to service a request. Activation entails the object being retrieved from the database and bound to an IDL interface skeleton so that the desired implementation's method can be executed.

Activation of objects is required, as it is inefficient and most likely impossible to have every object stored in a database instantiated in memory and registered with an ORB. Thus objects have to be passively stored until needed. Database adaptors must possess an activation mechanism to locate and retrieve objects to service a request.

The next complexity is how are persistent objects referenced given that they are dormant in a database and not actively registered with an ORB. The answer to this is to embed unique identification information that identifies persistent objects into CORBA object references. When a database adaptor receives a request, the adaptor can extract the identification information and use it to activate the relevant object. The requirement of unique identifiers to reference persistent objects is a constraint on the type of database that can be used with a database adaptor. Object-oriented databases(OODBs) support unique identifiers to objects, but this is not the case with tuples in the relational model.

The database adaptor has to save and load state data of a persistent object, therefore has to perform a mapping between the two different models. Again, OODBs are more suited to this than other database types as their objects can effectively be the persistent state with no conversion necessary.

On activation of an object, its persistent data has to be loaded into its implementation object. The implementation object then has to be attached to an IDL interface skeleton to process the request. The database adaptor has to provide this functionality. For OODBs, this is simply a matter of binding the persistent object to the interface and forwarding the request to the desired method of the persistent object.

It is the responsibility of the database adaptor to integrate the transactions of the Object Transaction service with the transaction mechanism that the database provides. Further, the locking of persistent objects is controlled by the concurrency service.

Database adaptors offer a neat solution to CORBA object persistence by basing all persistence functionality in the adaptor. The adaptor is the ideal element to place the ability to provide persistence to server objects. The adaptor is ideal due to its functionality of creating object references, thus can implicitly encase persistent identifier information within the reference. Also the adaptor is the first element on the server side to receive requests, therefore it can activate the relevant object to service the request. The result of this solution is that CORBA objects managed by database adaptors become Persistent CORBA objects as they are referenced persistently and are stored persistently. This also has the desirable feature that the persistence of objects is transparent to clients.

3.4.2 Object-oriented Database Adaptors

Object-oriented database adaptors(OODA) are well suited to providing persistence to CORBA objects. The integration of object-oriented databases(OODBs) into CORBA is commonly called their "synergy"[Amirb 97], as the merger provides great benefits to each technology. OODBs supply the following beneficial features to CORBA[Amirb 97] :-

- Concurrent access to a large number of persistent objects.
- Data recoverability.
- Guarantee of data integrity in the event of failure.
- Database transactions provide ACID properties(see section 2.4.5).
- Better management of server memory.

OODBs gain the following benefits from its integration with CORBA :-

- Extended heterogeneity, that is clients of the OODB can be implemented in any language and on any system that CORBA is ported to.
- Lower coupling between client and OODB, the client does not have knowledge of database object models and does not depend on OODB proprietary data transport mechanisms and interfaces.
- Clients are more lightweight.

OODBs are easily integrated into CORBA compared to other types of databases. The integration is easier owing to the similarity of the CORBA and object-oriented database models. Both are based on the object-oriented concept of an object that has state, behaviour and inheritance.

The characteristic of the OODB model that makes it particularly fitting over other database models is its intrinsic feature of object identifiers. Object identifiers(OIDs) uniquely reference a persistent object within a OODB. Thus, to identify a persistent object in the CORBA environment, the OID can be encoded into a string form and embedded within a CORBA object reference. Therefore, not only do OODAs support persistent CORBA objects, but then also support persistent object references. Further, these features provide a firm persistence foundation that CORBA sorely needs.

Activation and persistent object referencing is automated by the OODA, but it is still the responsibility of the application developer to write implementation objects and make them persistent. Again, with OODBs this process presents few difficulties as the implementation object will also be a persistent object.

Further information on integrating object-oriented databases with database adaptors can be found in [Vasu 94][Baker 97].

3.4.3 Why Object-Oriented Database Adaptors cannot be the only CORBA persistence mechanism

Object-oriented databases(OODBs) integrate well with database adaptors to provide persistence for CORBA. This integration is straightforward due the closeness of the OODB and CORBA object models. However, many corporate organisations are unwilling to invest in OODBs for various factors such as:-

• Immaturity	OODBs are relatively new, therefore have not been extensively proven in the corporate environment in terms of performance, reliability and scalability.
• Existing storage	Organisations will already have existing storage solutions that have

solutions	consumed much investment e.g. relational databases. Thus, would be unwilling to cast away existing storage technologies.
• Existing data in existing datastores	Corporations will have large amounts of data already stored in their datastores. To make this data accessible to a new OODA based system would require the data being re-modelled and translated to the OODB, which is a large complex project within itself. This would also have the consequence of making existing client application defunct, as they will be incompatible with the new system.

The resistance for corporations to move to the unproved technology of object-oriented databases will leave developers having to utilise their current storage solutions. Thus, many developers of CORBA systems will not have the luxury of having a similar storage model and a low impedance mismatch that exists between OODBs and CORBA.

3.4.4 Integration of Database Adaptors with Traditional Datastores

Traditional datastores such as relational, hierarchical databases and files are the mainstay of many corporate data solutions. If a corporation is to move towards an enterprise wide system that is based on CORBA, then the invaluable data stored in these datastores must be brought into the reach of CORBA applications. The method of persistent data access that is most fitting with the CORBA model is with database adaptors. The database adaptor's purpose is to instantiate CORBA objects that represent persistent data making the data accessible from the CORBA environment.

For a database adaptor to support a datastore, the datastore must provide the following facilities:-

- Unique identification information that is suitable to be embedded into CORBA object references.
- Retrieval/storage of identified data from/to the datastore.

For datastore types other than object-oriented databases, the provision of these facilities introduces extensive complexity.

Providing unique identifiers to allow the referencing of data is the most problematic for traditional databases. This is due to their models not containing the concept of unique identifiers for units of data.

For example, the relational model uses primary keys to represent references to tuples(i.e.rows) of a table, where a tuple is analogous to an object's state. A primary key is one or more attributes of a table, that together are unique for each tuple within the table. Thus, to reference tuples that are object's persistent states would require the tuple's primary keys to be encoded into object references. This would be an awkward integration, as actual attribute values would have to be encoded into object references. This would cause a dependency between an object reference and its object state, if the object's state change then the object reference would be invalid. These factors are implicit to the very nature of the relational model and do not mix naturally with object-oriented models.

The Epicentre model embodies additional problems to those of the relational model for creating object references. The Epicentre model and the Data Access & Exchange(DAE) interface uses a mixture of natural identifiers(analogous to primary keys) and direct references to reference entities. Further attribute types making up an entity natural identifier can also be a direct reference, which makes the encoding identifier information even more problematic. This is assuming that the DAE provides a form of external direct reference representation, which it does not as it is an implementation detail that is specific to the DAE implementation.

In addition to mapping unique identifier information to object references, the problem of mapping data to objects is a complex issue.

In the relational model, an object will rarely map to a single tuple of a single table owing to the normalisation process. The normalisation process[McFadden 94] breaks up an object into multiple tables, so that attributes (table columns) are atomic, all attributes are functionally dependent on their primary keys and there is no transitive dependency between non-key attributes. Thus, object attribute types such as complex types, aggregate(i.e. container of many data items) types and references will be mapped to multiple tables.

Consequently, more than one query is needed to fetch or set the data resident in multiple tables. The mapping between object types and their tables has to be defined in some way, so that SQL queries can be generated from the definition. Also, executing multiple queries to gather data from multiple tables is very performance intensive and inefficient, especially considering that usually only a small part of the data gathered will be read or manipulated.

Mapping Epicentre entities to objects is more straightforward if there is only a one-to-one mapping between the two models. The nature of Epicentre and its definition EXPRESS language is similar to object-orientation. Both paradigms have concepts of attributes grouped into information units, inheritance and relationships between information units. These factors have the effect of making mapping an Epicentre entity to an object fairly simple. The only problem with the mapping is coping with the many Epicentre extended types. These types would have to be mapped to objects with a defined IDL interface that would provide operations to manipulate the type's data.

Mapping data from traditional datastores to objects, and further into CORBA objects that can be managed by database adaptors is feasible in some respects. Instances of datastore constructs (e.g. tuples, DAE entities) can be the persistent state of objects given some processing needed for mapping. The disadvantage of this mapping is that it is a complex and performance deteriorating task as data has to be moved across the impedance mismatch. However, this is only considering non-referential attributes.

The foremost obstacle to integrating traditional datastores into CORBA via database adaptors is that of representing unique identification information (i.e. internal datastore references) in CORBA object references. As previously described, object-oriented database adaptors embed object identifiers (OIDs) into CORBA object references. Here, both forms of referencing have the same semantics in that a single value (i.e. the reference) points to a universally distinct object. As their semantics are the same, the integration of the two referencing types leads to no complexities in their use. This is not the case with datastore models based on referencing with attribute keys, as with the relational model. The semantics of attribute keys is of searching through all objects of the same type, for the object with the matching attributes. Thus, there is a dependency between references and objects. Further, to integrate an attribute key based reference into an object identifier based reference leads to complexities in the use of the reference.

These complexities are the result of the differing views of how objects are represented and what entails equality between objects in object-oriented and relational models. In object-oriented models every object is innately unique and objects are the same if their object identifiers are equal. In relational models, object (tuples) uniqueness is based on the value of their attributes, therefore two objects are the same if their keys match. These differing concepts cause a profound discrepancy in the use of the two reference types. For example, creating a copy of an object-oriented object is simply a matter of creating a new object and copying the attributes of the source object. Copying objects in the relational model is more difficult due to the fact that the object's attributes must differ for the newly created object to be distinct from the source object.

The integration of attribute key references into object references is conceivable, but impractical due to the semantics of their use. This mismatch in the semantics of referencing types would cause serious integrity problems and complexities for applications that utilise the references. Hence, to support datastores with such referencing type semantics with database adaptors is unrealistic.

Unique identification information is also problematic with logically defined data access interfaces such as DAE (section 2.6) and SDAI (section 2.5). The problem with data access interfaces of this type is that they are independent of any specific datastore implementation. Any references to objects/entities are specific to the datastore implementation, hence unique identification information contained in a reference is of an unknown format. It is also inaccessible and is not guaranteed to be unique outside of the client's session. Consequently, datastores of this type are also inappropriate for integration with database adaptors.

This discussion has only considered traditional datastores with already existing data models and data being integrated with database adaptors and subsequent accessibility from CORBA. It has been argued that this integration is far from ideal, if not impossible due to mapping problems and the merging of references of different semantic types. This does not entirely exclude traditional datastores from integration with datastore adaptors as some traditional datastores can be given an object-oriented wrapper. There are products[DBTools 98][ONTOS] available that give relational databases an object-oriented wrapper, so that from the application point of view the database appears similar to an object-oriented database. The problem with wrapping is that the relational tables have to be automatically generated from a data model by a tool provided by the wrapper product. The generated tables will be modeled in such a way as to incorporate object inheritance, aggregation and object identification. The consequence of this process is that the tables have to be of the wrapper's design and cannot incorporate already existing models. Therefore, it is viable to integrate database adaptors with relational databases given an object-oriented wrapper and no requirement to support existing data. The major disadvantage of this integration is its inefficient performance compared to using a pure object-oriented database. This inefficiency is caused by the time taken to execute the many queries required to fetch a single object.

3.4.5 The proposed OMG Persistent State Service

The OMG is currently considering submissions[PSS 98][PSS 98b] for a Persistent State Service(PSS)[PSS RFP]. PSS will replace the retired Persistent Object Service and will be based on a formalisation of the database adaptor approach to CORBA object persistence.

One of the primary problems that PSS will overcome is the binding together of components that comprise a CORBA persistent object. On activation of an object these components have to be created or retrieved and then bound together to service the request. The components of the PSS model are shown in figure 3.6. These components are described as follows:-

- Persistent Object Adaptor(POA) is the new OMG object adaptor standard. One of the new capabilities of the POA is the ability to embed and extract object identifier information inside CORBA object references. Currently, this ability is proprietary to specific ORBs.

- CORBA objects represent the persistent object IDL interface to external clients.
- Servants process requests from the CORBA objects. The servant's duty is to activate and bind to an actual application object.
- Application objects provide the implementation of the CORBA objects.
- Incarnations provide persistent state to application objects.
- Persistent objects are the actual persistent data resident in a persistent store. Incarnations are the cached form of this persistent data in memory.
- Persistent stores hold persistent data that represents persistent objects.

The PSS specification will provide interfaces and descriptions of how these components can be plugged together to provide persistent objects. It will also describe how persistent object identifier information can be embedded into object references and how it can be used to locate persistent objects. However, this persistent object identifier will simply be based on a single unique value. The specification will also describe how the service interacts with the object transaction service and how concurrency is achieved.

The final PSS specification will be based on the two proposal specifications. In studying

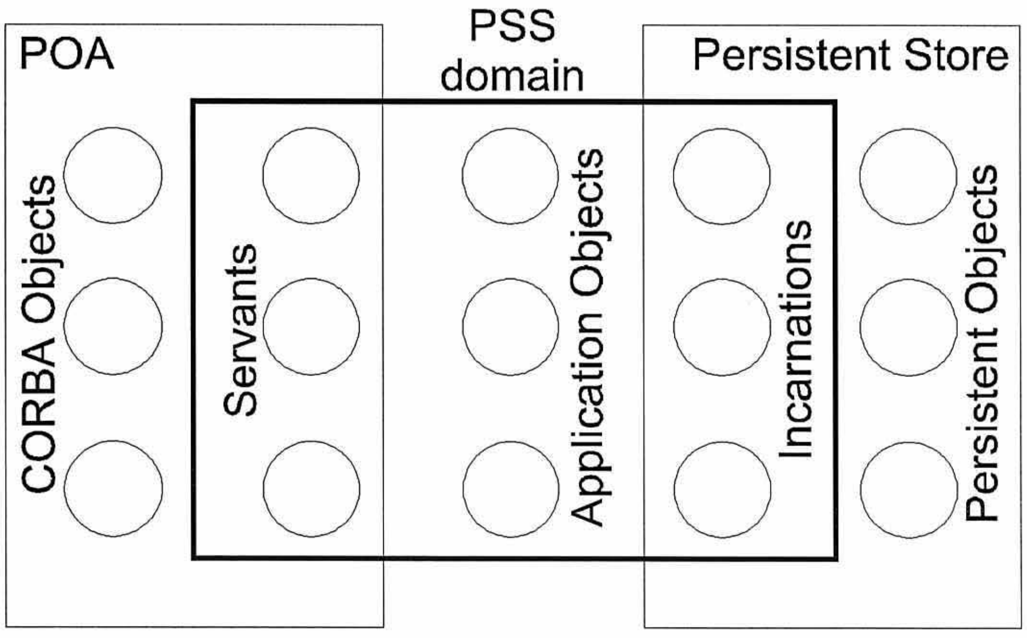


Figure 3.6: *Elements of persistent CORBA server supporting the Persistent State Service*

these proposals, the final PSS specification should be an excellent solution to CORBA persistence in a heterogeneous language and datastore environment. PSS will be a well designed and integrated service, unlike its predecessor - POS. However, the datastore model types that PSS can support still have to be close to that of the CORBA model.

Summary

This chapter has presented methods in which CORBA applications/objects can access persistent data that resides in heterogeneous datastores. These persistent data access techniques include: -

- Proprietary language-dependent data access interfaces e.g. I/O functions, database APIs.
- Standard IDL interface data access e.g. POS, SDAI.
- Persistent CORBA objects managed by database adaptors.

Proprietary data access interfaces entail an impedance mismatch due to the mapping needed to transfer data between logical and storage implementation models. To cross this impedance mismatch requires much conversion code that is bug prone, sensitive to model changes, time/cost consuming and has poor portability and reusability. Also there is the possibility of vendor lock i.e. becoming totally dependent on a single software product and its vendor. This situation does not resemble a component architecture that loosely couples components that are the basis of CORBA.

OpenSpirit is an excellent example of a CORBA application architecture that is segregated into three tiers. The middle logic layer provides components representing common E&P entities. These components rely on datastore wrappers to popular E&P datastores to provide persistence. The datastore wrappers map data between the OpenSpirit logical model of data and the datastore model of data. The author has personally helped implement the OpenSpirit to Epicentre mapping. This mapping had to be manually coded. This was possible due to the course granularity of the OpenSpirit model. For models of finer granularity, manually coding the mapping would quickly become too complex and bug-prone, thus automating the mapping code would become necessary. PrismTech's Mapping Manager Architecture is a possible technique of achieving this automation.

Standard IDL data access interfaces should be the primary techniques that CORBA applications use to access persistent data. The OMG Persistent Object Service(POS) has been discredited and subsequently retired due to major design faults and under-specification. One of the few beneficial features of POS was its characteristic of supporting a broad range of datastore model types, rather than a single datastore model i.e. object-oriented database model.

The STEP Standard Data Access Interface(SDAI) and its IDL binding provide a greatly missing functionality to the CORBA paradigm. Using the SDAI, CORBA applications can share and manipulate structured data that is resident in datastores. However, the SDAI specification is deeply rooted in STEP concepts and technologies, such as the EXPRESS data modeling language, EXPRESS data types and SDAI has no integration with CORBA services.

The persistent data access technique that is most fitting and consistent with the CORBA architecture and object model is the use of database adaptors. Database adaptors extend the functionality of the object adaptor by storing the persistent state of CORBA objects in a database. Database adaptors work well with datastore models similar to that of CORBA's object model, but for models outside the object-oriented paradigm (e.g. relational database model) the integration is problematic. The basis of these problems is the integration of unique identification information into CORBA object references due to inaccessibility and dissimilar semantics. Thus, object-oriented databases integrate elegantly with database adaptors, but not with other datastores supporting other types of models.

The Persistent State Service(PSS) is a formalisation of the database adaptor approach and is to become the future standard for CORBA persistence. PSS is the ideal technique of accessing persistent data by giving the data a CORBA object representation to the external CORBA environment.

Chapter 4

Analysis, Requirements and Design of a Persistent Data Access Service

4.1 Introduction

This chapter provides a short analysis of the current situation of persistent data access in CORBA, highlighting the problems with the currently available solutions. Given these problems, the requirements for a persistent data access service to solve these problems is presented. The majority of the chapter discusses high-level design considerations and solutions of the persistent data access service, so that an implementation of the service will meet the requirements put forward.

4.2 Problems Encountered from the Analysis of CORBA Persistent Data Access Solutions

The current CORBA persistent access solutions examined in the previous chapters have shown that access mechanisms to persistent data are lacking in the CORBA environment. This section provides an analysis of the functionality that is missing from these solutions as well as of the problems which a persistent data access solution should solve.

The most unsatisfactory problem with CORBA persistent data access is that CORBA has no standardised technique for persistent data access, hence leaving this area open for proprietary solutions. The introduction of proprietary solutions providing a functionality that is commonly used amongst applications is against the principles of the Object Management Architecture(OMA). The OMA adheres to the principle of component reuse that saves repetition of functionality and increases portability and reusability of components. Applications with proprietary data access solutions will not have these benefits.

Proprietary data access solutions inherently cause an impedance mismatch between the application and the data storage. Usually, it is the developer's duty to manually implement the mapping between the two models. This mapping is time/cost consuming, bug-prone, performance inefficient and sensitive to changes.

In directly using a proprietary data access solution, a developer has an additional software technology paradigm to learn, maintain, purchase, to be aware of technological advances and to consider implications of changes to applications. It would be beneficial to CORBA developers to put this data access functionality behind IDL interfaces, thus providing a great deal of insulation from the proprietary data access solutions.

For models of increased complexity and fine granularity, an automated approach to mapping has to be taken. The problem with mapping automation is the complexity of interacting with a proprietary data access interface, where every data entity, identifier information and data type has to be mapped to the target model. Thus, creating an automated mapper is a major development task that has to be justified. The complexity of such a development task is vastly increased by the many different data access interfaces that datastores support and as well as their many data definition models. Consequently, some degree of insulation is required from proprietary interfaces and models, so that mapping software is presented with the same interface and model for each datastore type.

Datastores with data models that are not similar to the object-oriented model are difficult to integrate into CORBA with the database adaptor approach. This difficulty is mainly caused by lack of unique identifiers for units of data. As a vast of amount of data are stored in datastores of this type, these datastores should be brought into reach of the CORBA environment.

Different datastore types have different types of internal datastore references. From the analysis in the previous chapter, there are basically two kinds of referencing:- attribute key referencing and direct referencing. Attribute key referencing indicates the referenced data unit(s) by matching attribute values held in the key with attributes of all data units of a particular type e.g. foreign key of a tuple. Direct referencing entails the reference holding some value that directly identifies the reference data unit e.g. a DAE handle to an Epicentre entity. Both forms of referencing should be supported.

For data access solutions to be reliable they should be integrated with the CORBA Transaction and Concurrency services. This is considered a critical part of a service dealing with persistent data and a factor that the previously reviewed persistent data access solutions(except for the future PSS) had not even considered.

Data manipulation mechanisms should be efficient. The persistent data access solutions discussed in the previous chapter are generally inefficient for certain types of data e.g. large arrays of data or manipulating large numbers of small objects. The inefficiency is due to data being kept in the datastore server process and is manipulated remotely using requests. To manipulate this kind of data requires many requests, the performance overhead caused by the requests will be intolerable for certain applications e.g. CAD. Therefore, server based persistence solutions such as database adaptors and consequently PSS will fail in this area. A more ideal approach for some situations would be the option to transfer the data in bulk to the client process and manipulate it locally i.e. caching data. Object-oriented databases typically cache data in client applications, but this leads to a high cohesion between application and datastore.

In connection with moving data in bulk, it should be possible to specify a sub-graph of data entities to be retrieved and stored as a single unit. This will increase the efficiency of application data manipulation as arbitrary collections of data can be moved back and forth between client and store in a single transfer.

Different data model types have a different set of data types, such as Epicentre extended data types and SQL time based types. A data access service would have to be flexible enough to allow manipulation of these data types.

It should be possible to give persistent data a CORBA object shell, allowing the manipulation of the data through an user provided IDL interfaces. Thus, application can access the persistent data through a static interface, rather than using the persistent data access service directly.

4.3 Requirements

In examining problems and missing functionality of current persistent data access solutions, the following set of requirements that a persistent data access service should supply is devised :-

- A set of services to allow manipulation of data stored in heterogeneous datastores.
- Achieve all manipulation using components supporting CORBA IDL interfaces.
- Allow data entities from differing data model types to be manipulated in the same manner.
- Avoid adding or changing any functionality of the datastore or the implemented data model so to prevent the operation of existing legacy applications. This will be of great benefit to the migration to new CORBA based systems as old and new systems can run in parallel.
- Avoid ORB vendor proprietary mechanisms.
- Allow the manipulation and navigation of datastore internal references, where references can be single unique values or attribute keys.
- Decrease the impedance mismatch between application and datastore models.
- Remove the responsibility for the implementation of mapping code from the application developer.
- Allow the efficient manipulation of data, whether caching locally or manipulating remotely.

- Design services in a component fashion to allow the plugging together and exchange of components that work together.
- Integrate with the CORBA Object Transaction and Concurrency services.
- Reuse and extend current CORBA services whenever possible, thus keeping to the Baushaus principle.
- Allow binding of CORBA objects to persistent data. This is a form of late binding(see SDAI, section 3.2.2). Without the bound interface to the data, early binding can be used to manipulate the data.
- The ability to retrieve and store sub-graphs of objects in a single access to a datastore.
- The flexibility to allow the handling of datastore data types that are not native CORBA data types e.g. SQL time based types, Epicentre spatial types.
- Minimum effort needed to map proprietary data definition models to the service's standard data definition model service.
- A query service to gain initial references to persistent data entities.

4.4 High-Level Design of a Persistent Data Access Service

This section provides an outline for the design of the services that will meet the requirements mentioned previously. It describes the major problems involved in constructing such a service and presents solutions to these problems and the components that will make up an implementation of the service. The behaviour and interaction of these components is presented along with the problems the components are a solution to.

In this section, persistent data will be referred to as data objects, due to the fact that data only has state and no behaviour unlike true object-oriented objects. Data objects can be any type of persistent data such as :-

- Variable sized arrays of binary data e.g. files, Binary Large Objects(BLObs).
- Units of information that have a set of typed attributes e.g. tuples, behaviour-less object-oriented database(OODB) objects, EXPRESS entities.

4.4.1 Manipulation of Data Objects

A persistent data access service will have to provide basic manipulation operations. These operations include creation and deletion of data objects as well as providing read and update operations of the data object's attributes. In addition to these operations, there is a need to control the storage of data objects i.e. retrieve and store operations that will pull the data object out of a datastore or push it back.

To manipulate data objects in a standardised manner requires an independent representation of the data object from its stored form(Figure 4.1). Thus, an impedance mismatch is created that requires mapping code to move data across. The purpose of mapping code is to issue 'get' and 'set' calls on the datastore interface to retrieve or store data between the two models. Examples of mapping code would be:- the execution of SQL statements to retrieve or set tuples and retrieval or setting of attributes of identified objects/entities via an OODB or DAE interface.

Mapping code can be generated manually or automatically. Ideally the automated approach should be utilised. The manual approach requires the application developer to manually code the mapping, which creates many problems (see OpenSpirit and its approach to data access). Tools for the automatic generation of mapping code for specific datastores will be more reliable, consistent, bug free and take the responsibility away from the developer.

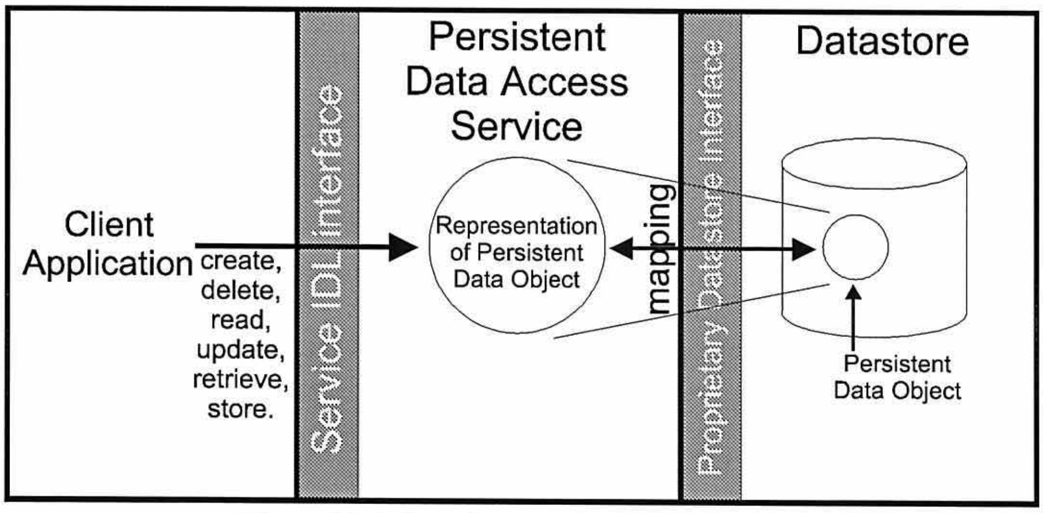

Client Application
create, delete, read, update, retrieve, store.
Service IDL Interface
Persistent Data Access Service
Representation of Persistent Data Object
mapping
Proprietary Datastore Interface
Datastore
Persistent Data Object

Figure 4.1: *Independent data object representation from its proprietary stored form*

Generally, there are two techniques for the generation of automated mapping code: static and dynamic generation(Figure 4.2). Static generation entails tools producing mapping code that provides routines for retrieving and storing each data object type involved in the model. The static mapping code is then linked with the service at compile time.

The dynamic generation of mapping code occurs at runtime with no pre-compiled code. Dynamic mapping code can generate retrieval and storing routines by examining meta-model information on each data object type that it handles. The meta-model information is provided by a facility where the structure of data object types can be queried i.e. the attribute types can be discovered and inheritance hierarchies can be traced. Therefore the mapping code will understand the structure of data objects it handles. Consequently it can generate the relevant 'set' and 'get' calls to retrieve/store data objects from/to their stored form.

Each type of mapping code has benefits and drawbacks. The static mapping generation is faster at runtime, but becomes increasing larger with larger sized data models. Dynamic mapping generation is slower at runtime, but does not increase in size for larger data models. Dynamic mapping also has the advantage that is resilient to changes in data models, while static mapping code will have to be regenerated with changes in data models.

Another consequence of static mapping is that there is a high dependency between tools, components, data models and programming languages implementing the static mapping.

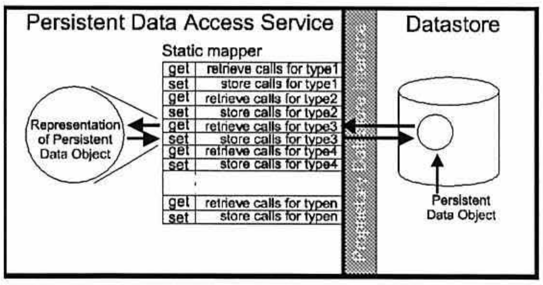

Persistent Data Access Service
Datastore
Static mapper
get retrieve calls for type1
set store calls for type1
get retrieve calls for type2
set store calls for type2
get retrieve calls for type3
set store calls for type3
get retrieve calls for type4
set store calls for type4
get retrieve calls for typen
set store calls for typen
Representation of Persistent Data Object
Persistent Data Object

Figure 4.2a: *Static mapping code*

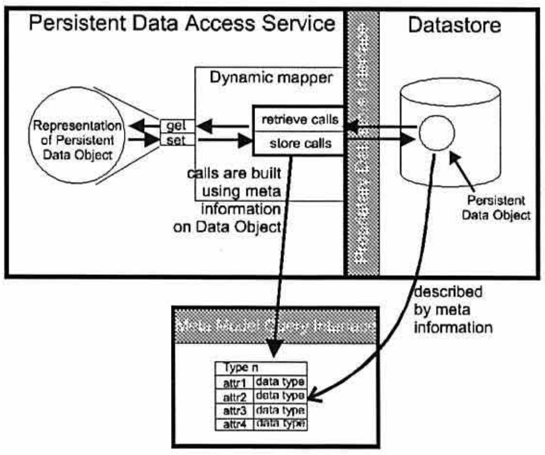

Persistent Data Access Service
Datastore
Dynamic mapper
Representation of Persistent Data Object
get
set
retrieve calls
store calls
calls are built using meta information on Data Object
Persistent Data Object
described by meta information
Meta Model Query Interface
Type n
attr1 data type
attr2 data type
attr3 data type
attr4 data type

Figure 4.2b: *Dynamic mapping code*

This high dependency is detrimental to scalability of systems and causes system parts to be non-reusable. On the other hand, the dynamic mapping with its meta-model facility allows generic components to be built that handle data objects regardless of their structure. Thus, there can be many components in a system reusing the meta-model facility to handle data objects. Lowering the dependency of components like this improves the scalability of systems.

Providing description information on data objects is a vital facility, therefore a meta-model facility is a critical component of a persistent data access service. Consequently, a meta-model facility should be a system-wide service in its own right.

4.4.2 The Meta-Schema Model

The structure of data objects is described within a schema/data model. In the following sections the terms 'data model' and 'schema' will be used interchangeably. A schema describes the attribute structure of data objects, relationships between data objects, the inheritance hierarchy of data objects and any user-defined data types. A schema is defined in a Data Definition Language(DDL). There are many different forms of DDL such as SQL DDL, ODMG ODL, ISO EXPRESS, XML or even programming language based DDLs such as C++ classes, Java classes. Each DDL has its own data definition model that specifies language constructs, data types, relationship semantics, inheritance, behavioural and constraint definitions. However, if only the data definition parts of these models are considered, ignoring behavioural and constraint parts, then there is a great deal of commonality between these models.

The difference between data models and data definition models is further clarified for an improved understanding of the following material. A data model is a description of data that represents entities from the problem domain that the application is concerned with. A data definition model is the form that this description takes. The data definition model will define the way in which entities are represented and related.

A requirement of the persistent data access service is the manipulation of data objects in a common manner, regardless of their stored form and their native data definition model. Thus, to manipulate data objects in this way requires a common data definition model to be presented to applications. This standard data definition model will transcend the proprietary data definition models of datastore types as it will be generic enough to allow the mapping of all proprietary DDL schemata to it(Figure 4.3). A standard data definition model is possible due to the fact that proprietary data definition models are all based on a similar structure of data objects, attributes, data types, relationships and inheritance.

The standard data definition model will not be another data modelling facility with an associated data definition language. It is simply a standardised method of describing data defined in proprietary data definition models to applications, thus providing the application with a single logical view of the structure of data.

The standard data definition model that will be used to describe the structure of data

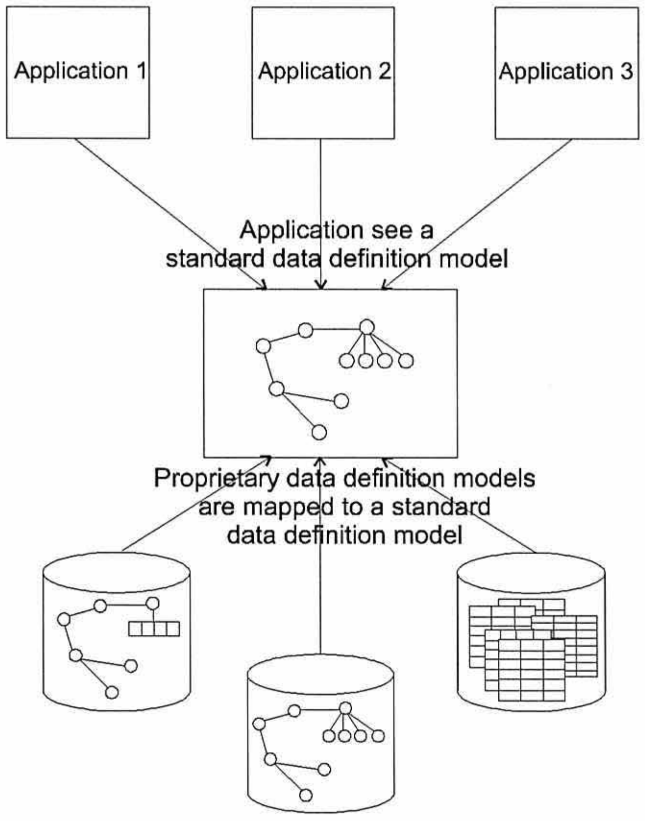

Application 1
Application 2
Application 3
Application see a standard data definition model
Proprietary data definition models are mapped to a standard data definition model

Figure 4.3: *Proprietary data definition models are mapped to a standard definition models*

objects manipulated by the service is shown in figure 4.4. The model is presented as an object model. A set of instances of the object model will provide meta-information that describes a data model. Consequently the object model is called the meta-schema model as it describes schema/data models.

The meta-schema model was designed by carrying out an in-depth analysis of several data definition languages including SQL, EXPRESS, ODL. The analysis examined the features of each language and extracted a model of the common features and the relationships between these features. The examination produced an initial rough design of the object model. After many redesign iterations and re-examination of the data definition models and languages, the final model was completed.

An overview of the object model is explained in the following:-

- A Schema object contains many schema defined types, many schema constants and many entities.
- The Schema Defined Type is specialised into sub-types that allows the definition of user defined types that are defined globally in the schema.
- Schema constants provide the definition of global constants in the schema.
- Entities are the principal information element of a schema. The model permits the definition of an entity's parent and child entities, thus specifying its inheritance hierarchy. An entity contains many attributes.
- The Attribute object defines its name and type of an attribute of an entity.
- The Type object is the supertype of all the objects that define the type of an attribute. The possible types of an attribute are Entity Reference, Aggregate, Any, Base and Defined.
- An Entity Reference defines the attribute as being a relationship between entities. The Entity Reference defines the type of the referred entity and whether the reference is an inverse reference. If so, then the forward referencing attribute can be specified.
- Entity Reference is further specialised into Absolute and Key references. This specialisation allows the model to incorporate the two types of referencing used in data

definition models. These referencing types are pointer based referencing as in object-oriented models and referencing by attribute values as in the relational model.

- Aggregate and its sub-types of Array, Bag, List and Set provide types to represent an attribute with multiple elements e.g. a set of references to other entities, a bag of integers.
- Some data definition models have an Any type, which allows a value of any other type to be assigned to it.
- The Base type is the parent of basic literal types and fixed arrays of them.
- The Defined type allows the attribute type to be defined as one of the schema defined

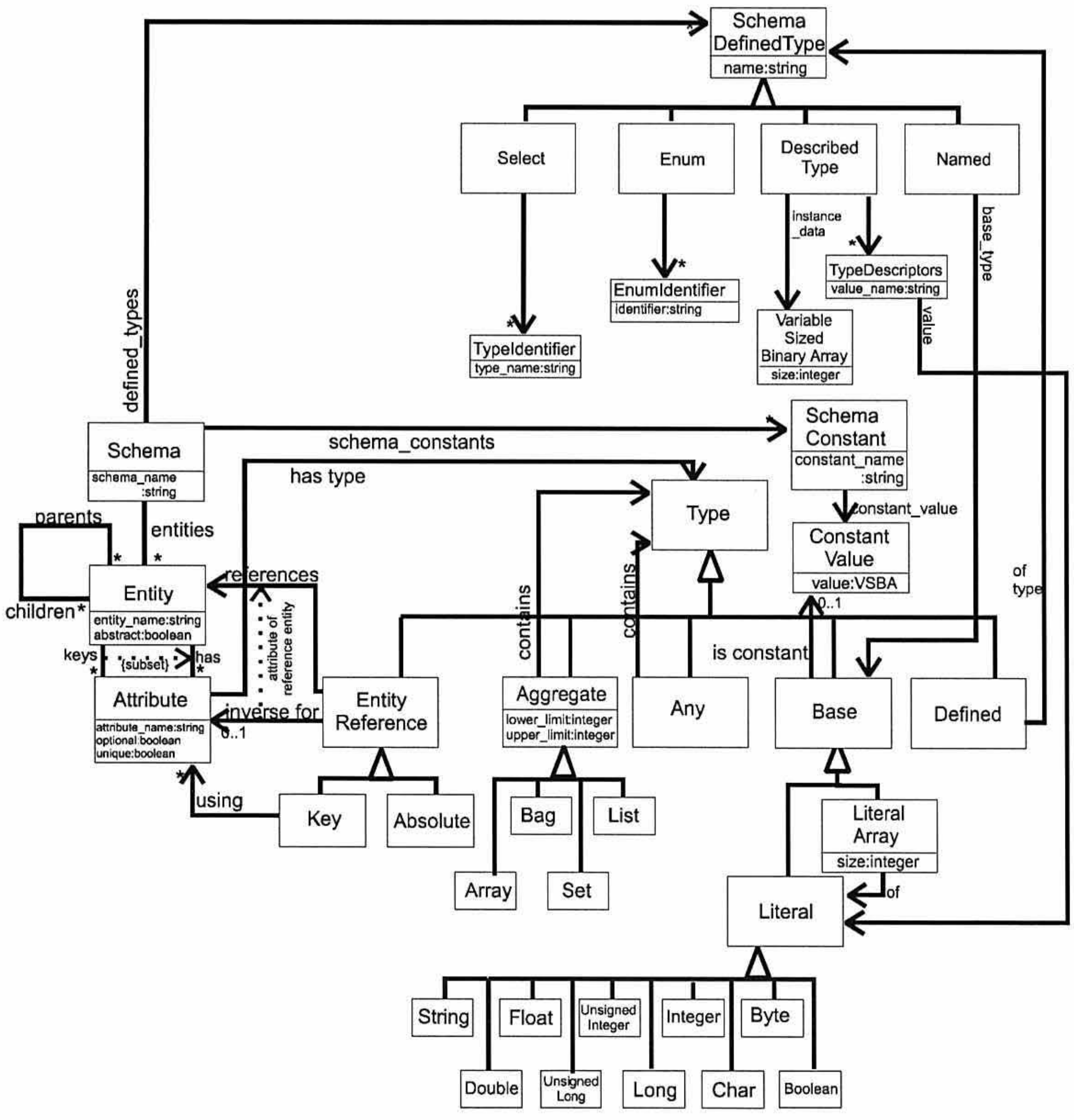


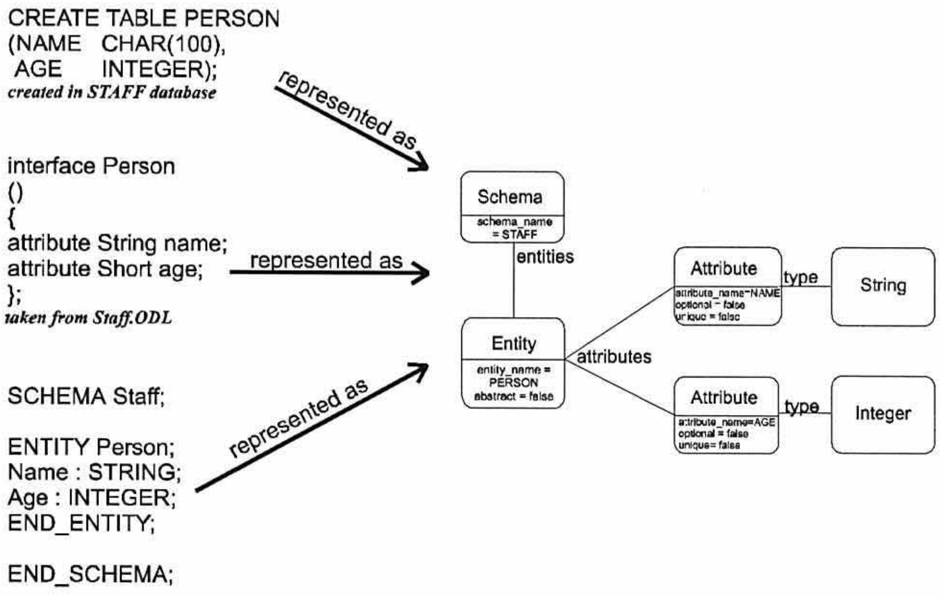
Figure 4.4: *The Meta-Schema Model*

types :- Select, Enum, Described Type, Named.

- Select allows the value to be any of the identified types.
- Enum provides the enumeration of a list of identifiers.
- The Described Type is provided for the purpose of representing types that are exclusive to a proprietary data definition model. For example, SQL has time based types, and the Epicentre model has over 300 extended types. The Described Type can therefore be used to describe a type and associate the type's instance data with it. It is the responsibility of the application to read the describe identifiers and handle the instance data in a consistent manner with the structure of the data.
- The named type simply gives a Base type a name such as used in the 'C' language 'typedef' statement.

The meta-schema model can represent the majority of features that data definition models provide. Not all the features provided in the meta-schema model will be used by proprietary data definition models. For example, relational models will not use the inheritance hierarchy feature or the Schema Defined Type facility. Object-Oriented based models can make use of all features provided.

The meta-schema model is the data definition model of data objects manipulated by the service. Proprietary data definition models of datastore types will map their schema



```
CREATE TABLE PERSON
(NAME  CHAR(100),
 AGE    INTEGER);
```
created in STAFF database

```
interface Person
()
{
attribute String name;
attribute Short age;
};
```
taken from Staff.ODL

```
SCHEMA Staff;

ENTITY Person;
Name : STRING;
Age : INTEGER;
END_ENTITY;

END_SCHEMA;
```

Figure 4.5: *Proprietary schema definitions are represented as instances of the meta-schema model*

definitions to the meta-schema model(Figure 4.5). Client applications will therefore manipulate data objects that the structure of which is described by instances of the model. It may be necessary for application components to query the structure of the data objects, allowing the component to dynamically handle and manipulate data objects. To provide the meta-information to components, the meta-schema model has to be implemented in a run-time form that permits the querying of the model instances. This is achieved by representing each meta-schema object as a CORBA object. Therefore it is necessary to give each meta-schema object an IDL interface allowing the querying of its attributes. The IDL interfaces for the meta-schema model can be found in appendix D. Components implementing these interfaces will supply the meta-model facility mentioned in the previous section(4.4.1). This facility will allow components to find out the structure of data objects via the meta-schema objects allowing the manipulation and mapping of data objects(Figure 4.6).

4.4.3 Local and Remote Data Object Manipulation

This section discusses the advantages and disadvantages of local and remote manipulation of data objects and the design decision that the persistent data access service should support both forms. The primarily problem with local manipulation is moving data across the distributed environment to the client application. The solution is to split the service into

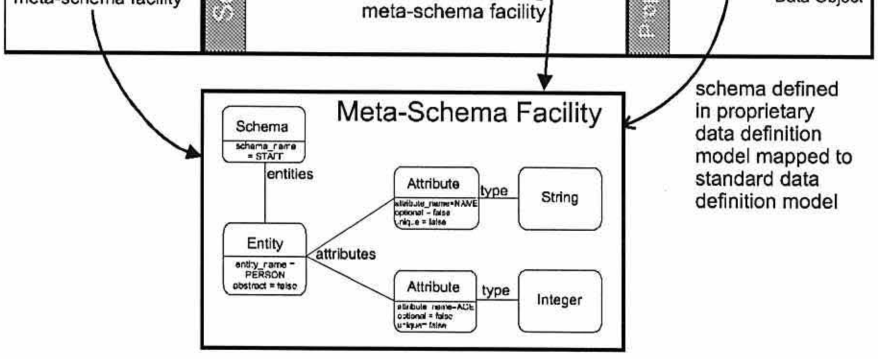


Figure 4.6: *Meta-Schema Facility comprises of CORBA object that are instances of the meta-schema model classes*

client and server components connected by a stream-based mechanism to transport data.

4.4.3.1 Data Manipulation Models

The efficiency of a system that is manipulating persistent data is very dependent upon the location of the data. There are two models that are typical of the location of data that is being manipulated (Figure 4.7). In the first model, data is kept local to the datastore server process and all data manipulation is carried out by the server on behalf of the client. This model is characteristic of relational databases. The second model involves the moving of data from the datastore server process to the client process where the client directly manipulates the data using locals calls. Thus, the client is caching the data locally for fast repetitive manipulation. This model is typically used in object-oriented databases and file systems.

The optimum model to use for data manipulation is primarily dependent on the frequency that the data objects are accessed. For the manipulation of data objects that are infrequently accessed, it is more efficient to manipulate them in the server, as the overhead in transferring the data objects to the local process would outweigh the benefits of fast local manipulation. Data objects that have a high possibility of frequent access are more efficient if cached in the client's local process, thus gaining from the performance of virtual instantaneous local function calls compared to slow remote procedure calls. Local caching is also more efficient for very large data objects or very large numbers of data objects. In these situations the data objects can be transferred in bulk to the local process. A specific

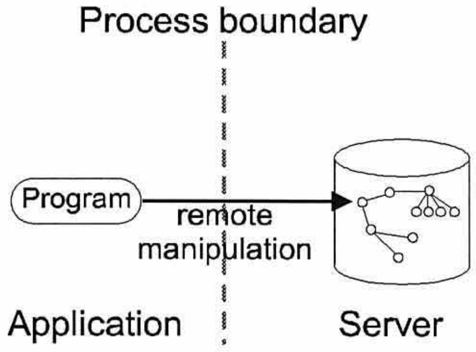


Figure 4.7a: *Data is remotely manipulated*

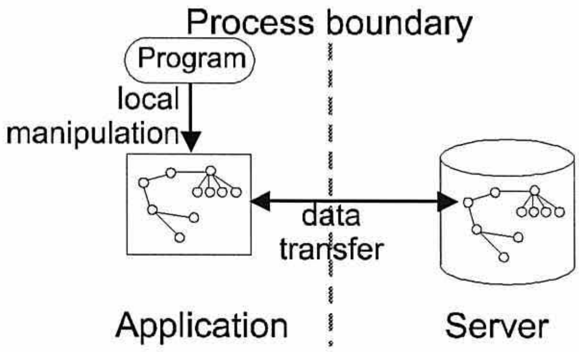


Figure 4.7b: *Data transfer to the local process and manipulated locally*

example of where local caching of large numbers of data objects is in CAD applications. CAD drawings contain enormous numbers of data objects such as points and shapes. These data objects are also frequently accessed, therefore it is a necessity that these data objects are cached.

Each type of datastore is usually heavily biased towards one of these data manipulation models, thus certain application types are more efficient in one type of datastore than another. The persistent data access service is providing CORBA wrappers to datastores. Consequently the manipulation model that the datastore types uses is irrelevant to CORBA applications. But the decision of the manipulation model that the actual service uses has to be taken. The ideal situation is for the service to allow for both models, leaving the decision to the application of which method to use.

4.4.3.2 Design Factors for Implementing Local and Remote Data Object Manipulation

Manipulating data objects locally or remotely is of no concern to actual CORBA application code. This is because of CORBA's location transparency characteristic. A request on a local CORBA object is initiated the same way as a request on a remote CORBA object. As a result, it makes little difference to the application which manipulation model is used, except for performance related factors. Therefore, the only design factors involved in supporting both manipulation models is how the application indicates the location that it wishes to manipulate data objects and how the service will transfer the data objects to that location.

The first problem is how applications can decide the location that data objects will be manipulated in. This can be solved by creating the component(s) implementing the service in the relevant location i.e. in the server or application process. The facility that covers component creation in CORBA is the LifeCycle service (see section 2.4.2). Therefore, the components defined in LifeCycle service can be used to create an instance of the service. A LifeCycle factoryfinder component can be used to locate a factory object capable of creating an instance of the service. The location of the factory selected will determine where the data objects are manipulated (Figure 4.8).

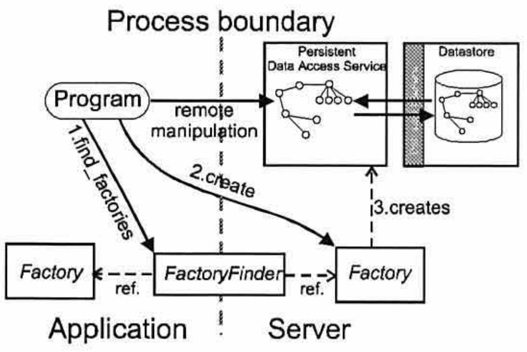


Figure 4.8a: *A remote factory is used to create an instance of the service for remote manipulation*

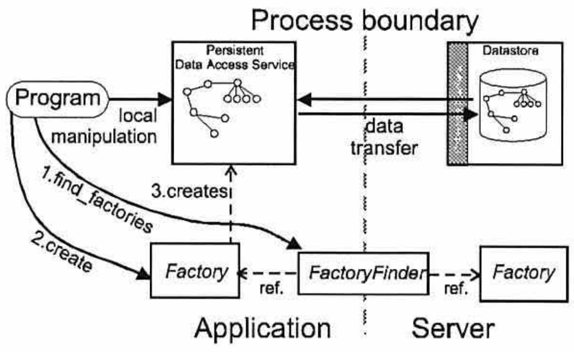


Figure 4.8b: *A local factory is used to create an instance of the service for local manipulation*

The second problem is specific to the model of caching of data objects in the client application. The problem concerns how data is transferred between processes in the distributed environment. The most obvious solution is to make use of the data transfer mechanism of the datastore interface. The problem with this solution is that it is introducing a high degree of coupling between the application and datastore interface software. The coupling constrains the application to the vendor, platform, language as the datastore interface library would have to be linked to the application. Also, another constraint might be a limited number of available licenses for the datastore, hence only allowing a certain number of clients to be connected simultaneously. These constraining factors are exactly what must be avoided.

A better solution would be a generic client side to the service that uses an independent

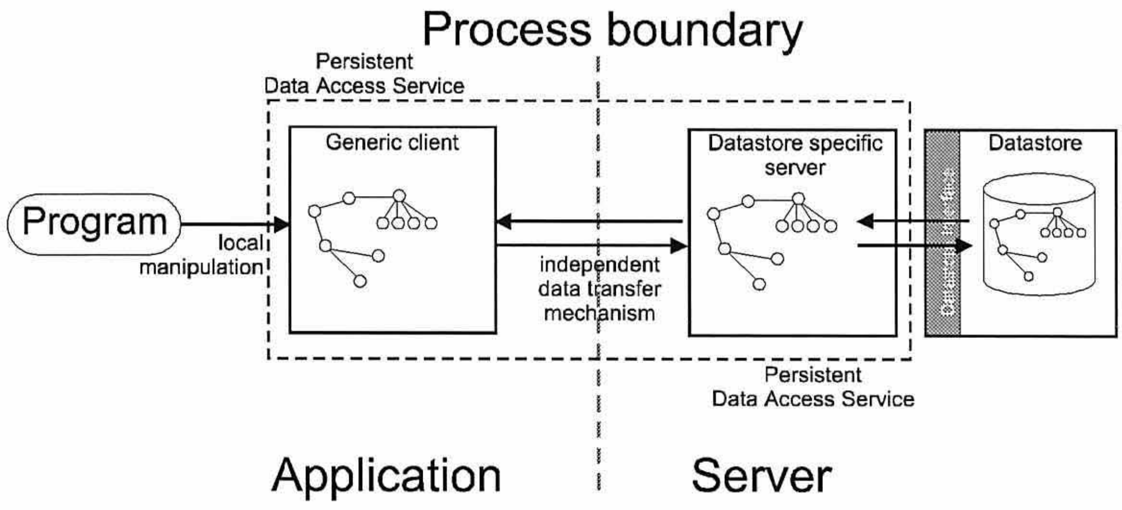

Process boundary
Persistent Data Access Service
Generic client
Datastore specific server
Datastore
Program
local manipulation
independent data transfer mechanism
Persistent Data Access Service
Application
Server

Figure 4.9: *A generic client side to the service using a data transfer mechanism that is independent of the datastore data transfer mechanism*

communication mechanism to transfer data to and from the server side(Figure 4.9). The server side component will be dependent on the datastore software, but at least the client application will be a pure CORBA application with no dependencies on proprietary communication mechanisms.

4.4.3.3 Caching Data using Streams

To cache data objects in the client application, the data objects have to be moved from the server across the distributed environment to the client. The critical problem with this solution is that CORBA is being used as a middleware layer and the essential requirement for inter-operating using CORBA is the definition of an IDL interface. This is problematic as data objects manipulated by the service will not have IDL descriptions, therefore cannot be passed as parameters of requests. Hence, data objects cannot be moved using normal ORB mechanisms.

One possibility is to use a dynamic data transfer method based on IDL's 'Any' type such as the Persistent Object Service's Dynamic Data Object protocol[OMG COSS]. But this would be grossly inefficient and does not offer a very convenient and concise interface.

An ideal solution would be to use a stream-based data transfer mechanism. A stream is a sequential array of bytes. Using a stream, data objects can be broken out into their composing basic literal values and written sequentially to the stream. The stream can then be moved to another location via the ORB and the data objects rebuilt by reading the data out of the stream(Figure 4.10).

Setting up a stream between client and server components of the service not only offers the benefit of transporting non-IDL defined data, but also has the benefit of being able to efficiently transport data in bulk. Using a stream, graphs of large numbers of data objects be can written to a single stream and moved in a single transfer and recreated at the other end(Figure 4.11).

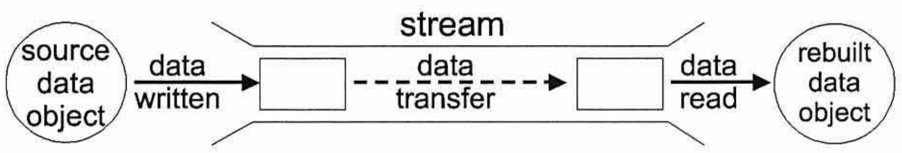


Figure 4.10: *Using a stream, data objects can be written to the stream, transported to their destination and rebuilt there*

The functionality offered by streams is of great use to CORBA systems as it provides CORBA with the ability to copy by value i.e. take a copy of an object, move it else where and recreate. Currently, CORBA only offers copy by reference, where only references to objects can be passed in request calls. Hence, the streaming ability is providing a missing functionality for CORBA and therefore should be designed as a service in its own right. Future CORBA standards will include an extension to the ORB and IDL to support an IDL 'valuetype', which declares an object that can be passed by value. But it does not offer the powerful ability to serialise of graphs of objects.

The design for a streaming service will have to define and describe how the following functions are carried out:-

- A stream is set-up.
- Data is read in and out of the stream.
- Data is transported between ends of the stream.
- Data objects are stored and rebuilt.
- Graphs of related data objects are stored and rebuilt.

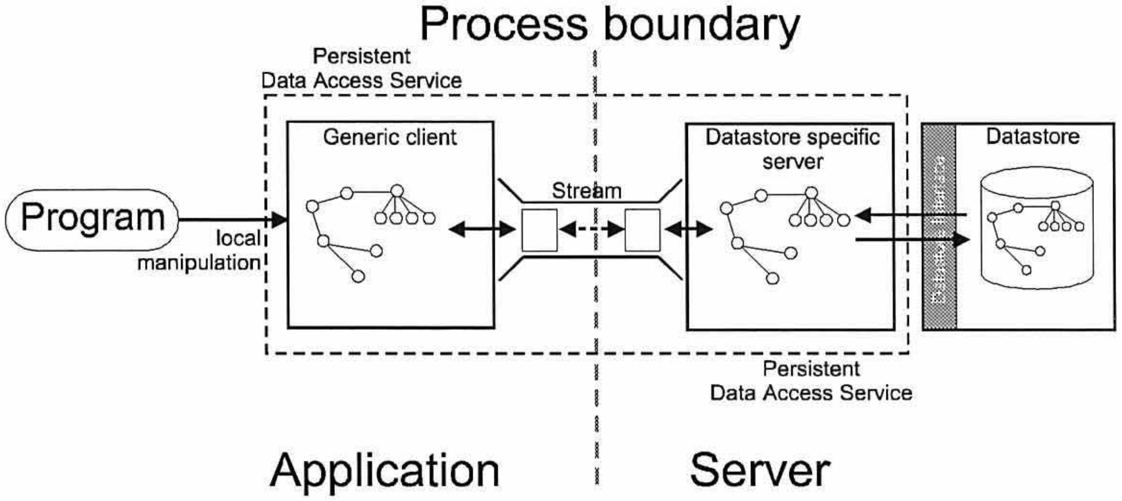


Figure 4.11: *The stream can be used to efficiently transport large numbers of data objects between client and server parts of the service*

The CORBA Externalization service(see section 2.4.3) already provides some of this functionality. The Externalization service defines IDL interfaces to write and read basic literal types in and out of a stream. The service also depends on the LifeCycle service for the creation of objects that are read out of the stream.

As for graphs of objects, the Extenalization service specifies the Relationship service for the compound serialising of CORBA objects. However, the Relationship service defines the relationships between CORBA object instances. Data objects are not CORBA objects, therefore this functionality is of no use.

As a result, a standard stream format to support the storage streamed graphs of related data objects will have to be devised. This format will have to keep the structure of the data objects and internal data object references in a consistent form for recreation in the destination. An analogy can be drawn between this and STEP files(part 21), that describes how EXPRESS instance data can be stored in files. In this case, streams are being used instead of static files and the primary use of the stream is data transportation rather than data storage.

A component that serialises data objects to or from a stream must have the capability to perform the following :-

- discover the structure of the data objects.
- access the local representation of the data objects.
- (de)serialise the data objects that conform to the standard stream format.

The serialising component must cross the impedance mismatch between local representation and the standard stream format representation. Thus, the component can benefit from the meta-schema facility to discover the structure of data objects and to perform dynamic mapping between the representations(Figure 4.12).

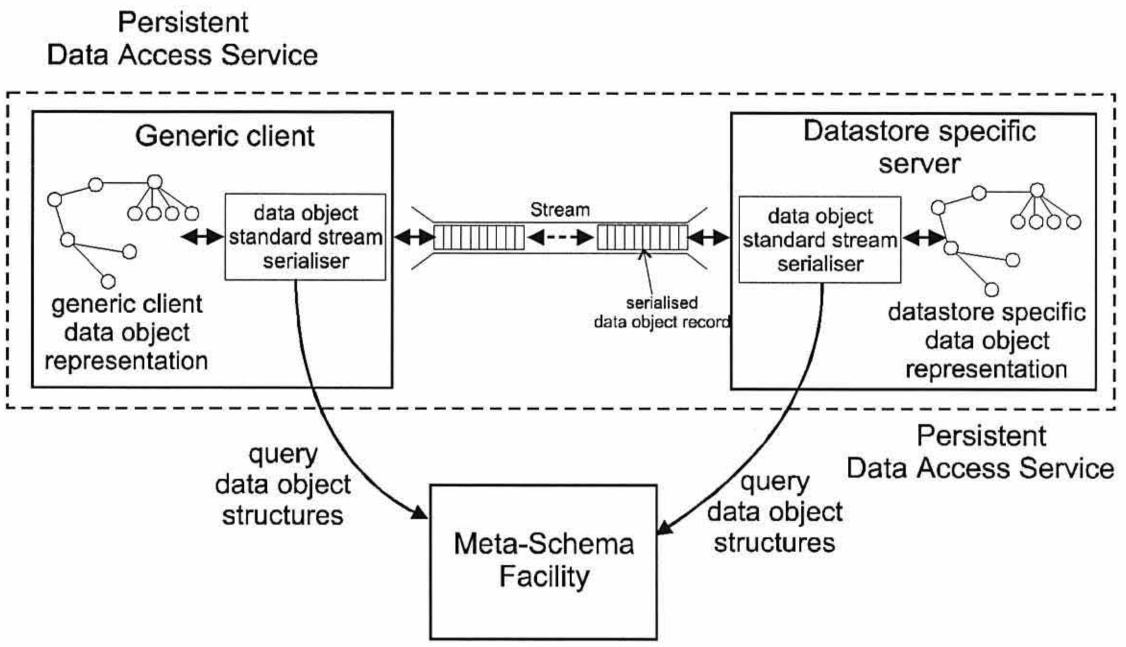

Persistent
Data Access Service
Generic client
data object standard stream serialiser
generic client data object representation
Stream
serialised data object record
Datastore specific server
data object standard stream serialiser
datastore specific data object representation
query data object structures
Meta-Schema Facility
query data object structures
Persistent
Data Access Service

Figure 4.12: *The meta-schema facility can be used to discover the structure of data objects enabling dynamic (de)serialising*

It should be emphasised that the combination of stream transport across the distributed environment and a standard serialised format for structured groups of data objects is a powerful general tool for the exchange of data. In this case, it is being used as a data bus between client and server parts of the persistent data object service.

4.4.4 Sessions

Client applications that access persistent data commonly perform its manipulation within the context of a session. A session is a logical connection between the client and the datastore server. The session offers the client a virtual exclusive access interface to the datastore, hence hiding the concurrent access of other clients. Through the session, all datastore specific operations are performed:- opening/closing of logical collections of data, manipulation of data, creation/deletion of data, execution of queries, access control, accessing meta-data, applying locks and beginning/ending transactions.

The persistent data access service will reflect this model of access through a session to provide a logical connection to the proprietary datastore session and as an access point to the rest of the service's functionality. Thus, using the session interface, a client can gain other interfaces of the service that provide the functionality to perform the datastore specific operations mentioned previously.

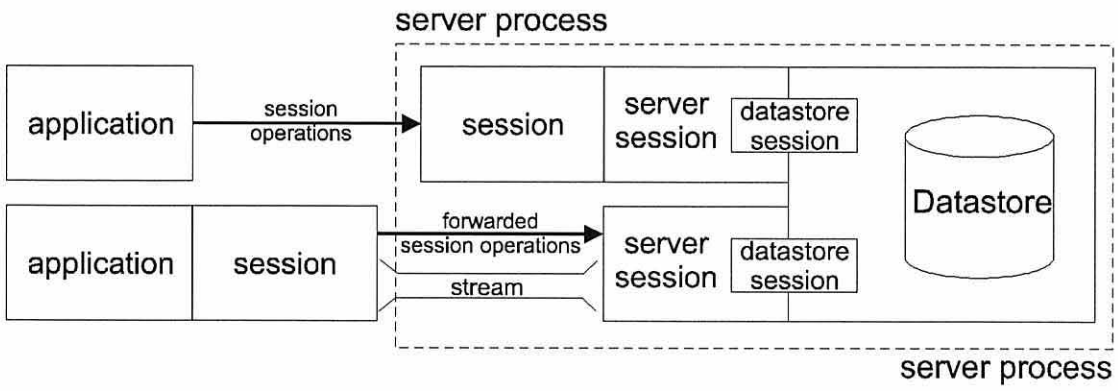


Figure 4.13: *Configurations of client and server sessions to provide for local or remote data manipulation*

A design decision of the persistent data access service is to support the two location manipulation models i.e. local or remote data object manipulation. The result of this decision is that the service is implicitly split into client and server parts. Consequently, a single logical session is implemented with a client and server session components. As with streams, the client session should be a generic implementation that is independent of datastore specific software. The client session is also the component where actual datastore independent representations of data objects are manipulated. The server session on the other hand is specific to the datastore and supplies a bridge between the service's session operations and the datastore proprietary session(Figure 4.13). Also, for each session created in the server process, a new thread of execution is created for the session to run on, hence there are multiple sessions running in parallel accessing the datastore.

The client session is the component that permits the application to access and manipulate data objects. Therefore, the location of the client session determines the data manipulation model in use. If the client session is remote from the server session, then it is the client session's responsibility to set-up a stream between it and the server session.

4.4.5 Integration with the Transaction and Concurrency Services

Transactions and locking in proprietary datastores are generally controlled by the client application from within a session, using the datastore interface(Figure 4.14a). However, transactions in the distributed CORBA environment do not have this simplistic model of a direct transaction between client and server. CORBA transactions are distributed. In this model a single distributed transaction can cover many requests amongst many CORBA objects(Figure 4.14b). Consequently, the client of the datastore may not be the originator and controller of the transaction.

CORBA transactions are provided by the Object Transaction Service(OTS)[OMG COSS](see section 2.4.5). The OTS interfaces are used to create and start a distributed transaction by some component in the CORBA environment. This component is the originator of the transaction, thus it begins and ends(i.e. commits or rollbacks) the transaction. The originator component will make requests on CORBA objects. These objects in turn might make requests on other objects etc. A chain of requests that has been initiated from within the context of a transaction is unfolded (Figure 4.15). Any objects in this chain that are transaction-aware will implicitly pass transaction information onto objects they are making requests of. The state of certain objects in the chain might be affected by whether the transaction is committed or not. Frequently, these affected objects will be supporting some sort of persistent storage, so that when the transaction is committed the state change can be written to the store. These objects who's state is dependent on the transaction outcome are called resources.

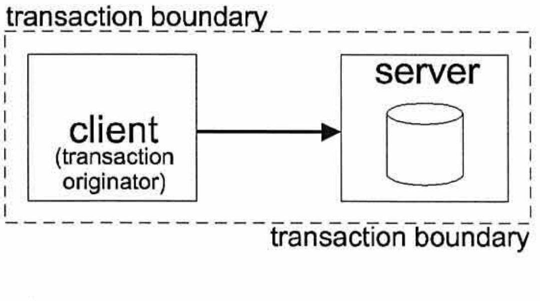


Figure 4.14a: *Direct transaction between client and datastore*

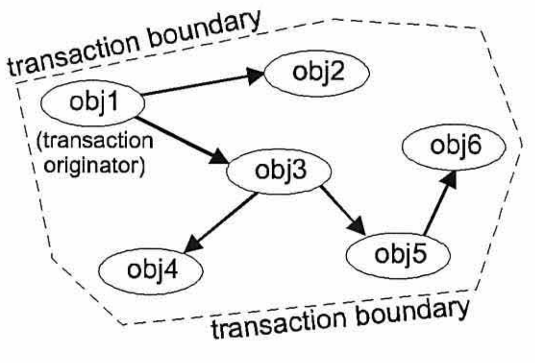


Figure 4.14b: *Transaction is distributed covering many components*

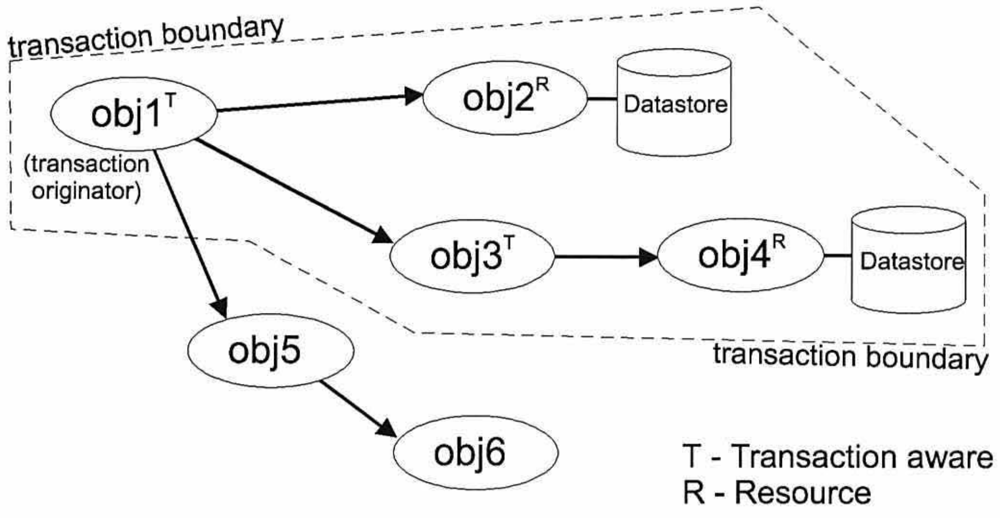


Figure 4.15: *A chain of requests within a transaction. Transaction aware objects convey transactions information, Resources take part in completing a transaciton*

Each component involved in the transaction has the right to force a rollback, but only the transaction originator can initiate a commit. The change of state of all resources within a transaction is dependent on how the transaction is completed. If the transaction is completed with a commit, then a two phase commit protocol must be performed, so that each resource agrees to the commit. To be involved in the two phase commit each resource must implement the OTS Resource interface. For an object to be transaction aware it must inherit the OTS TransactionalObject interface.

A requirement of the persistent data access service is its integration with the object transaction service, therefore it must fit in with the OTS transaction model and support its interfaces. As it happens, the design of the session model presented in the previous section integrates neatly into the OTS transaction model and interfaces. In the session model, the client session is the application's gateway to the service's functionality and the server session is the bridge between the datastore and the service. The client session is typical of a transaction-aware object that is only concerned with whether its own operations have been correctly executed and with passing transaction information on to other objects. The server session is a resource object as it has data that has to be persistently stored upon commit, and has to vote in the two-phase commit whether it can commit or not. Thus, the session components can implement the relevant OTS interfaces and therefore take part in the distributed transaction(Figure 4.16).

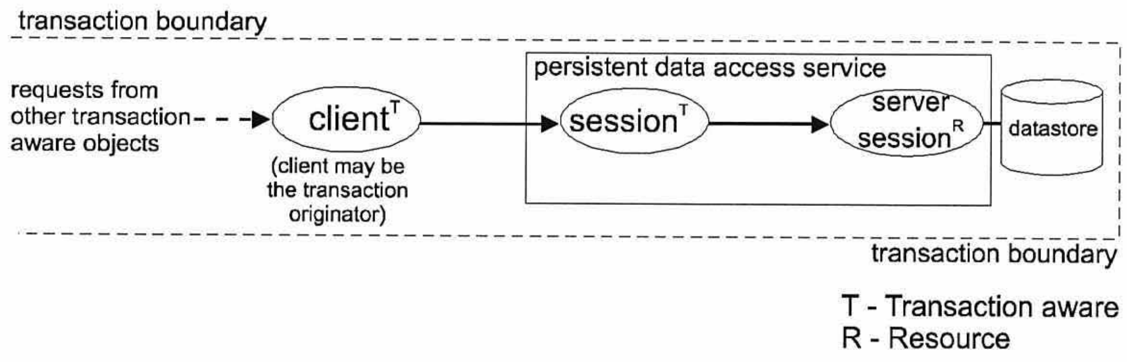

transaction boundary
requests from other transaction-aware objects
client^T
(client may be the transaction originator)
persistent data access service
session^T
server session^R
datastore
transaction boundary
T - Transaction aware
R - Resource

Figure 4.16: *The client session is transaction aware and the server session is a resource*

Datastores are usually being concurrently accessed, therefore locking is necessary to prevent interference between clients and to maintain the integrity of data and the system as a whole. Hence, it is necessary build locking functionality into the design of the service. The CORBA Concurrency Control Service(CCS)[OMG COSS](see section 2.4.5) supplies standard interfaces for clients to gain locks of different types on resources/CORBA objects. CCS provides the *Lockset* interface that allows locks to be gained, dropped or upgraded. CCS also provides the *LockCoordinator* interface that permits the OTS to drop any locks held on a resource at the end of a transaction.

Again, the session components can implement the relevant CSS interfaces(Figure 4.17). Both client and server session can support the *Lockset* interface. The client session will simply forward locking requests to the server session where the locks will be held. The server session will implement the *LockCoordinator* interface enabling the OTS to explicitly drop its locks upon the completion of a transaction.

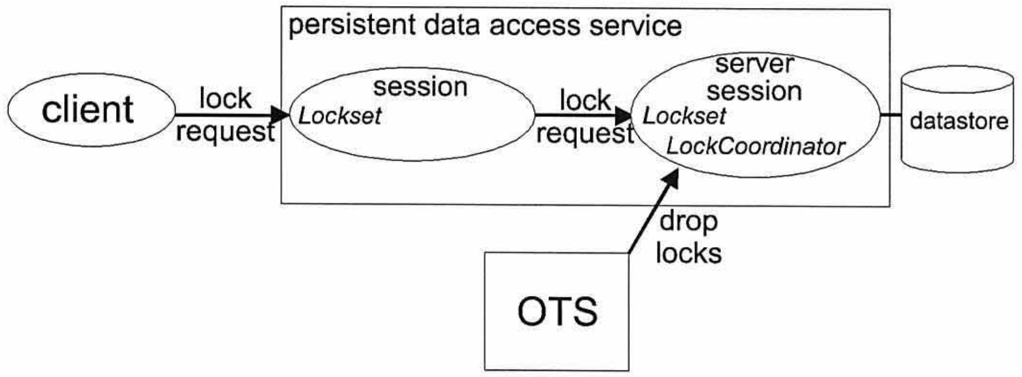


Figure 4.17: *The session components will support the relevant Concurrency Control service interfaces*

The session components are the primary elements to integrate transaction and concurrency services. The session components accomplish this integration by inheriting the relevant interfaces and implementing the behaviour expected of these interfaces. The interface hierarchy of the session components is shown in figure 4.18.

The session components are divided into client and server side varieties. They further specialise into non-transactional and transactional types. Every session type inherits the *Lockset* interface, thus providing the locking operations. The transactional sessions inherit the necessary OTS interfaces, with the transactional server session additionally inheriting the *LockCoordinator* interface. The sessions are set-up in client-server pairs, where a pair depends upon whether the session will work within a transaction or not.

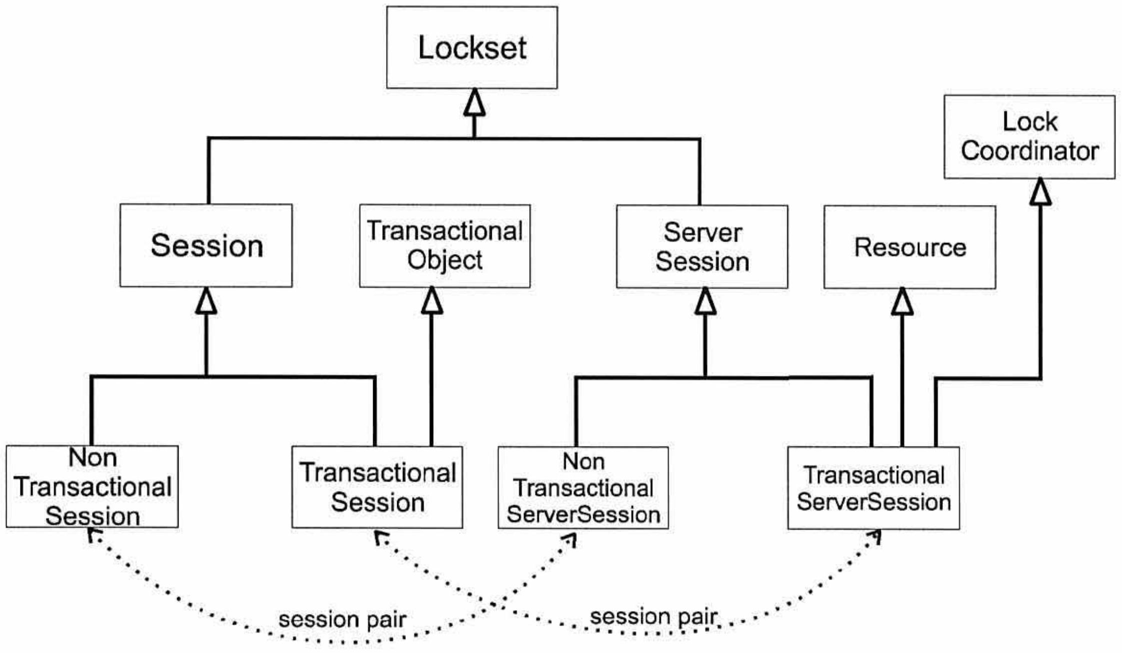


Figure 4.18: *Interface hierarchy of the session components incorporating transaction and concurrency service interfaces*

4.5.6 Representing Internal Datastore References

A facility that makes structured data defined in schema definitions so powerful is their property of representing relationships between data objects. A relationship is defined in data definition models as a reference attribute of a data object. A value of a reference attribute indicates the other data object(s) in the relationship.

Representing references is the most problematic property to deal with when manipulating data outside its native domain i.e. copying data out from a datastore to an external representation, manipulating it, then updating the original stored data. Moving non-referential attribute types between its native format and an external format is usually just a case of a one-to-one mapping of a native type to an external type. But for referential attribute types the mapping is far more complicated, mainly due to it being rarely possible to directly represent a reference in two domains. Direct representation is not possible owing to the many diverse methods to represent references, such as memory pointers, objects/structures containing referential information, attribute keys, object identifiers. These methods of maintaining references is specific to the native datastore and/or the programming language used to access the datastore. Therefore, a more complex method must be employed to map native references to external references.

Manipulating references in an external format entails further complexity issues. The integrity of references must be maintained, so that references copied out from the datastore still logically point to the same data object. Integrity of references must also be maintained when entities are updated in the datastore. Another complexity is that references have an associated type that constrains the type of data objects that can be assigned to it. A further complexity of reference types is dealing with the data object type's inheritance hierarchy in that any sub-types of the reference type can be assigned to it.

The design of the persistent data access service must address these problems of handling references. The initial problem is of representing datastore references external to the datasstore. This requires some standard form of storage technology independent representation. An ideal solution that is independent of storage technology and their implementation languages is for the server session to assign every data object retrieved from a datastore an object identifier(oid). Where the object identifier is an integer value that is unique within the logical session. Thus, data objects active in the service can refer to each other by their oid's(Figure 4.19).

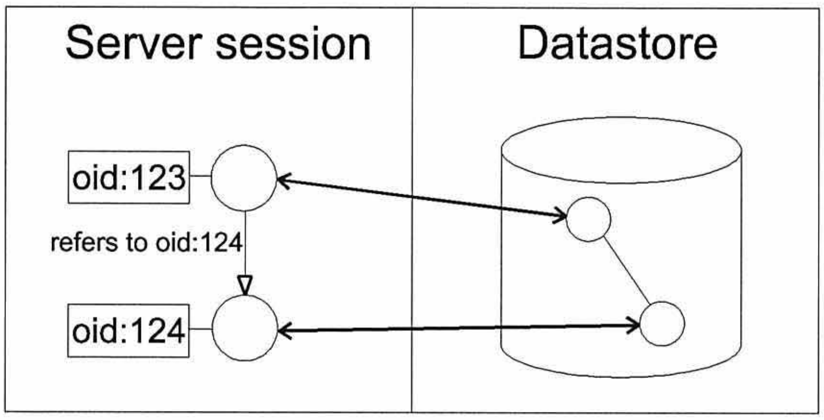


Figure 4.19: *Active data objects are given a session unique integer value providing them with an object identifier(oid)*

The benefit of assigning each retrieved data object an oid, is that data objects can be moved between client and server sessions via the stream and their relationships can be maintained. This is due to the service no longer being constrained by the datastore/language specific references. However, the server session has to be carefully planned to prevent inconsistencies between logical references via oid's and the native datastore references. One difficulty in maintaining consistency is that the server session must ensure that distinct data objects will have only a single oid within a session. Therefore, when a data object is retrieved from a datastore, the current list of active data objects must be checked to see whether the data object has previously been retrieved and assigned an oid.

It should be noted that this proposal to use object identifiers(oid's) is different from the object-oriented database(OODB) concept of object identifiers. An OODB oid is a permanent identifier for the lifetime of an object. Here, an oid is only valid for an object for the duration of a session as oid's are dynamically associated with objects upon retrieval. The OODB oid on the other hand will be the same for every session in which a particular object is retrieved. The significance of this is that OODB oid's can be dependably used in an external environment to the OODB, which is why they can be integrated into CORBA object references via object database adaptors(see section 3.4.2). The same sort of integration could be achieved with session dependent oid's, but the CORBA object references would only be valid for the duration of the session. Also, all objects referred to would have to be active in memory, unlike OODBs where objects are usually dormant.

The use of object identifiers is a mechanism that allows the persistent data access service to maintain the consistency of references between its own representation of references and the native datastore references. Nevertheless, this functionality is internal to the workings of the service and is hidden from the application.

The application view of references relates to the reference types of the meta-schema model. Applications will view references as attribute instances of meta-schema Key or Absolute types(see section 4.4.2). Applications can navigate a relationship by explicitly retrieving the data object by its attribute reference. If the attribute reference does not have an oid associated with it then the client session will ask the server session to retrieve the data object(Figure 4.20a). Once retrieved the data object will be associated with an oid, transferred to the client session and the oid associated with the attribute reference(Figure 4.20b). This technique is called swizzling[Vadaparty 95] and is commonly used with object-oriented databases.

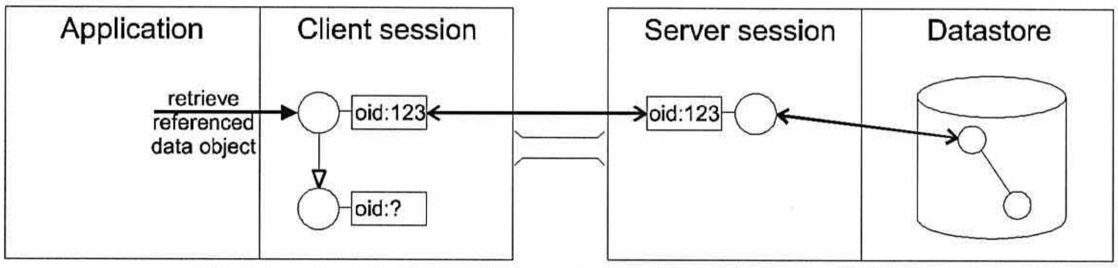


Figure 4.20a: *Attribute reference does not have an associated oid, requiring the referenced data object to be retrieved*

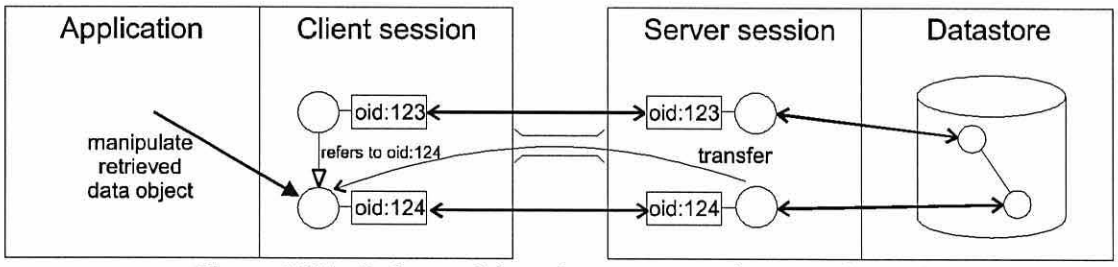


Figure 4.20b: *Referenced data object is retrieved, assigned an oid and oid is associated with attribute reference*

4.5.7 Querying Data Objects

A query in general database terminology means not only to select, but also to carry out general data manipulation operations such as creation, updating and deletion. For the persistent data access service, querying through the service is restricted to selecting data objects only.

The querying facility will permit applications to specify a select query, execute it and return a collection of data objects that meet the criteria of the predicates defined in the query. This enables applications to gain initial data objects whose relationships can be further navigated or to simply gain a collection of data objects that are of interest to the application.

The next design consideration is what will the structure of query statements be and how they will be executed. There is a large variety of query languages amongst the database community. The majority are based on standard languages of SQL[Vander 93] or OQL[ODMG 93], with many languages being proprietary versions of these. Hence, datastores supported by the service will have a diverse range of query languages. Thus, a simple query language is required that can be used to specify select statements. These statements can be translated to each native query language.

The query language will be based on the "select from <type> where <predicates>" statement. The <type> specifies the name of a entity type declared in a meta-schema model and the <predicates> provides evaluation expressions on attributes of the entity type or entities with a relationship to the specified entity type. Note that the returned elements are always data objects, so there is no need to specify what is passed back as in SQL select statements.

The next consideration is how to execute these statements. The natural solution to this is to examine integrating the CORBA query service into the service. Fortunately, the CORBA query service is generic enough to serve our purposes. The query service defines a *QueryEvaluator* interface that supports the submission of a query statement in a string form and returns an IDL 'Any' type containing the results. The implementation of the *QueryEvaluator* has the duty of translating a select query to the native language, executing the query on the local datastore and

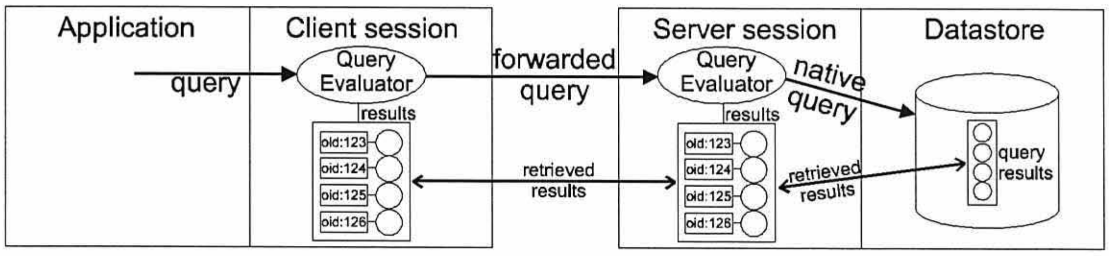


Figure 4.21: *Retrieving data objects using a QueryEvaluator*

retrieving the data objects indicated by the results of the query(Figure 4.21).

4.5.8 Retrieving Groups of Data Objects

A common programming pattern emerges when manipulating data objects, where the data objects are related by complex networks of relationships. This pattern involves the following series of steps :-

1. Execute a search for an initial data object.
2. Retrieve a related data object that is of interest.
3. For each retrieved data object, repeat the previous step for each relationship of interest.
4. Manipulate retrieved data objects.
5. Store any update data objects.

The effect of following this pattern is that a network of retrieved data objects is constructed in memory. Typically, each application type will repeatedly traverse the same set of relationships between data objects for each execution of the application.

However, using this pattern requires each data object to be explicitly retrieved from the datastore using separate retrieval calls. This is technique is inefficient. A superior approach would be to specify an initial data object and for all the relationships to be automatically traversed and the data objects retrieved. This approach has the benefit that all data objects of interest can be transferred in bulk to the client, thus being more efficient than multiple retrieval calls and data objects transferred individually.

To implement such a facility, the relationships that are traversed and the data objects that are to be retrieved have to be described. This description is called a Data Object Retrieval Map. The retrieval map has to somehow be presented to the system, so that the server session can automatically traverse relationships in the map so that an application can specify a retrieval map. The presentation of a retrieval map to the system can be conveyed in a form similar to that of the meta-schema model(see section 4.4.2). Because the retrieval map information is represented as CORBA objects with defined interfaces, the components of the system can examine these objects to find out the map's structure.

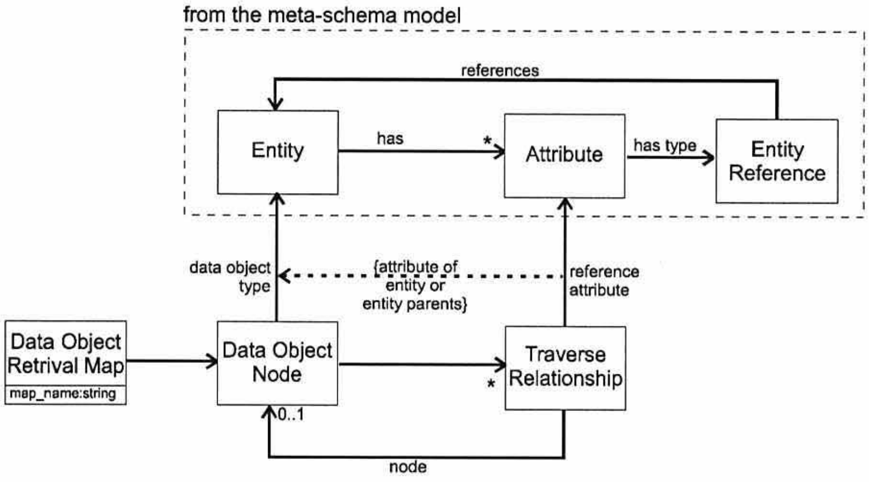


Figure 4.22: *The data object retrieval model*

The Data Object Retrieval Model is shown in figure 4.22. Instances of the Data Object Node and Traverse Relationship classes reference the entities and attribute references that make up a retrieval map. Each Traverse Relationship can reference another Data Object Node allowing multi-level retrieval of data objects.

Figure 4.23 shows an example schema of entities and relationships in EXPRESS. Using this schema, the following relationships could be traversed:- A.ref1->B and A.ref2->C.ref3->D. The instances of the retrieval model describing this traversal is shown in figure 4.24.

The Data Object Retrieval Map and its representation model is only a facility to improve performance for applications, thus is not vital to the persistent data access service. But traversing relationships from an initial data object is a common procedure that was also realised in the implementation of the OpenSpirit to Epicentre mapping(see section 3.2.2). Being able to automatically retrieve data objects in accordance with a set retrieval map is not only performance efficient, but also removes some work from the programmer.

```
ENTITY A;           ENTITY C;
 ref1:B;             ref3:D;
 ref2:C;            END_ENTITY;
END_ENTITY;
                    ENTITY D;
ENTITY B;           END_ENTITY;
END_ENTITY;
```

Figure 4.23: *An example EXPRESS schema*

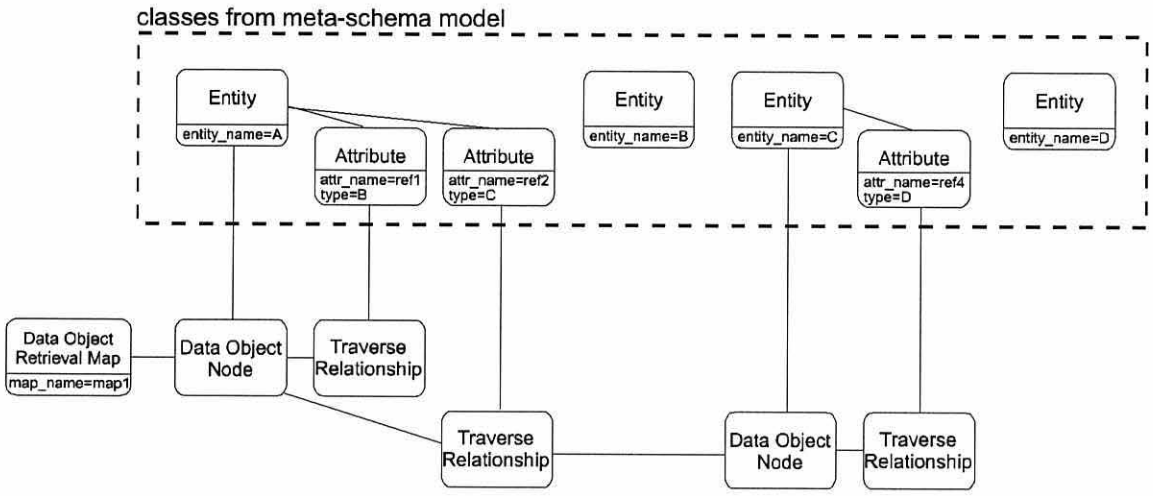


Figure 4.24: *Instances of the retrieval model describing the relationships of entities that will automatically be traversed*

4.5.9 Binding CORBA Objects to Data Objects

The persistent data access service is not dealing with the storage of CORBA objects, but with data objects, which are units of information that have no IDL interface and no behaviour. The storage of data objects is delegated to datastores. The service interacts with these datastores to manipulate their stored data objects. As data objects are not CORBA objects, the service must dynamically manipulate data objects by understanding their structure by comprehending their meta-information.

An application to manipulate data object attributes is provided with a single interface for every data object type. This interface allows applications to dynamically 'get' or 'set' attributes by providing the following information:- handle to a target data object, an attribute string name and typed variable. See figure 4.25 for pseudo code showing an example of dynamic and static manipulation. The disadvantages of dynamic 'get' and 'set' operations is that type checking of operations cannot be carried out at compile time, applications are more complex, hence more bug prone.

```
DataObject handle = ....;
AnyType value = "Mikey";

set(handle,"NAME",value);
```

Figure 4.25a: *Dynamic manipulation*

```
Person person1 = ....;
String name = "Mikey";

person1->set_name(name);
```

Figure 4.25b: *Static manipulation*

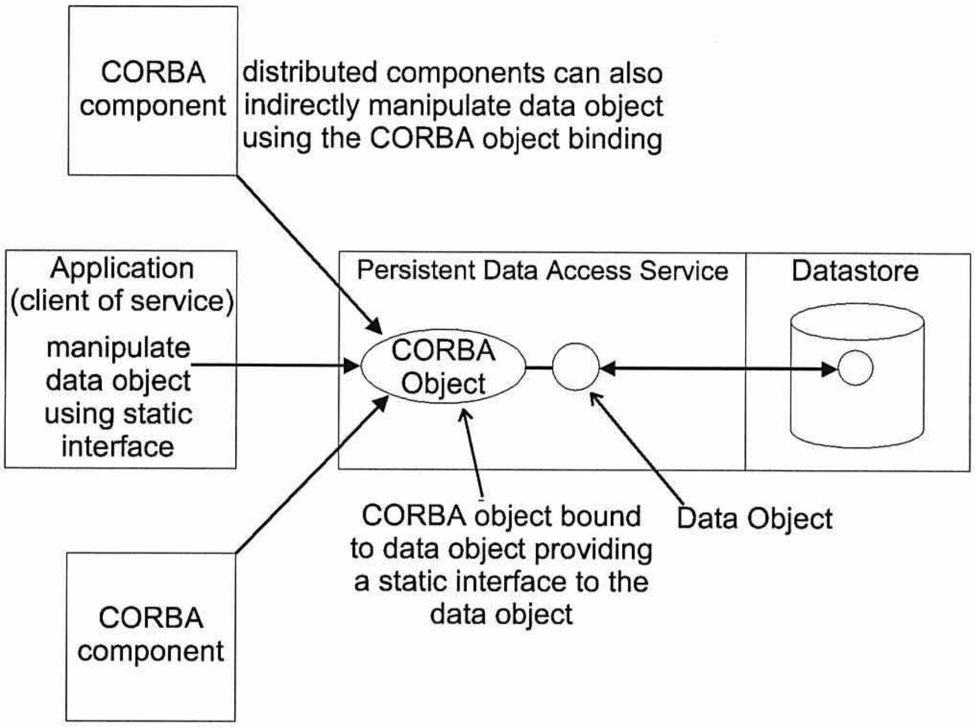

CORBA component
distributed components can also indirectly manipulate data object using the CORBA object binding
Application (client of service)
manipulate data object using static interface
Persistent Data Access Service
Datastore
CORBA Object
CORBA object bound to data object providing a static interface to the data object
Data Object
CORBA component

Figure 4.26: *CORBA object is bound to a data object providing a static interface for manipulation and allowing access from the distributed CORBA environment*

It would be advantageous to provide data objects with a static type-specific interface that would permit static manipulation. Further, this interface binding would naturally take the form of CORBA objects, thus providing applications with a static IDL interface to data objects. As well as having the benefit of a static binding, it would also have the benefit of making data objects available for manipulation from any component in the CORBA environment(Figure 4.26).

CORBA objects providing a static interface to data objects will have to be implemented by developers or be automatically generated from the meta-schema description by special purpose tools. A similar situation can be found in [Sauder 97], that describes an implementation of the STEP IDL SDAI. Here, CORBA objects are automatically generated from EXPRESS schemata and are used as wrappers for database objects.

Given a set of CORBA objects that behave as wrappers to data objects, the service will have to provide a mechanism to bind the two together. Given a handle to a data object of a certain type, its matching CORBA object wrapper has to be obtained. The basis of this problem is associating data object types with their appropriate CORBA object wrappers and instantiating the wrapper.

This functionality is provided by an integration of the CORBA Trading and FactoryFinder interfaces. In this integration, factories of the wrapper objects can register the data object types they support with the trading service. The persistent data access service can use the FactoryFinder object to return the relevant wrapper factory that can be used to create an instance of a wrapper object to act as interface to a data object(Figure 4.27). The integrated Trading and FactoryFinder service have also been suggested in the CosLifeCycle service specification.

This mechanism of binding of CORBA objects to persistent data objects is effectively providing a form of persistence for CORBA objects. However, the persistent data of the CORBA object is determined by the data object's schema rather than by the IDL interface attributes.

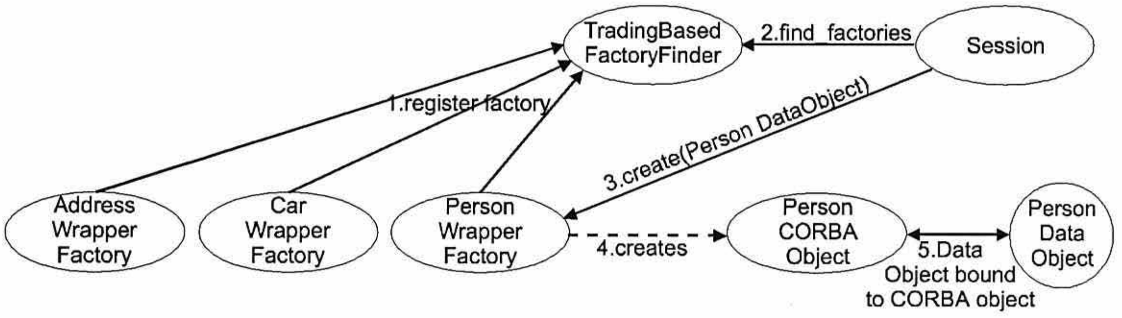


Figure 4.27: *Requests required to find and instantiate a CORBA object wrapper to bind to a data object*

4.6 Summary

This chapter has presented an analysis of problems with current CORBA persistent data access solutions. This primarily pointed out that there is no standard persistent data access service for CORBA, resulting in proprietary solutions. Consequently a high dependency is created with the storage solution. The future Persistent State service will mostly fill this functionality, but is only ideal for datastores that have models similar to object-orientation. Also, for database adaptor based solutions, all manipulation of data is carried out in the server, therefore a slow remote procedure call is required each time data is accessed. In some situations, the impact of this performance decline is unacceptable.

The outcome of the analysis is the formalisation of a set of requirements that the persistent data access service should meet. The highlights of these requirements are that the service should provide a set of IDL interfaces that will allow the common manipulation of data stored in heterogeneous datastores. The service should not impede existing applications accessing the datastore and integrate with the CORBA transaction and concurrency service.

The chapter continues by covering the problems involved in meeting these requirements and design decisions of how these problems can be solved. In the service, persistent data are called data objects as they only have state and no behaviour. To manipulate data objects in a common manner, the service must hold data objects in a form that is independent of the proprietary datastore representation, therefore mapping is required to transform data objects between the two forms. This mapping can be dynamically carried out by discovering the structure of data objects via meta-information.

To manipulate data objects in a common manner, applications must also have a common view of the structure of data objects. This view is provided by the meta-schema model. The meta-schema model is an object model. Instances of the object model will describe the structure of data objects. This description is provided by proprietary data definition models being mapped to the meta-schema model.

The next design decision is for the service to support both local and remote data manipulation models. This allows applications to cache data locally permitting fast repetitive manipulation. Providing this feature inherently splits the service into client and server components, therefore for local manipulation, a stream mechanism is required to transport data between the components. A stream mechanism provides the missing copy-by-value functionality to CORBA, and should be designed as a service in its own right.

Clients of datastores usually operate through a session. The session provides all the functionality of the data access interface as well as locking and transaction ability. This session model will be reflected in the service. A single logical session comprises of a client session and a server session supplying the client and server components of the service.

Integration of the transaction and concurrency services is critical to any reliable service. The service can combine with this functionality by implementing the necessary interfaces. These interfaces can easily be supported by the client and server sessions. The client session supports the TransactionalObject interface and the server session supports the Resource interface, allowing it to take part in transaction completion. Both sessions support the Lockset interface permitting the clients to specify the locking to use for the session. The server session additionally supports the LockCoordinator interface that allows the transaction service to drop any locks held.

Other facilities provided by the service are:- giving active data objects an session dependent object identifier to represent internal datastore references, a Query service interface for executing simple 'select' queries, a facility for specifying relationships that can be traversed for automatic retrieval of groups of data objects and the ability to bind CORBA objects supporting static IDL interfaces to data objects.

These design decisions presented in this chapter have laid out the basic components and facilities of the persistent data access service. In doing so, they have answered many questions of how the requirements specified at the beginning of the chapter can be met.

Chapter 5

The Stream Tunnel Service

5.1 Introduction

The Persistent Data Access Service(PDAS) requires a mechanism to transport non-IDL defined data across the distributed environment. This data transport mechanism is provided by the Stream Tunnel Service(STS)[Ball 98][Ball 99].

The purpose of the Stream Tunnel Service is to provide a mechanism to set-up and use distributed streams. Using the stream, data can be written to one end of the stream, transported to the other end of the stream and read out there. The benefit of this approach of transporting data in streams over using ORB request calls is that the structure of the transported data does not have to be statically defined in IDL operation definitions. This makes streams ideal for transporting data objects held in datastores and defined using the datastore specific data definition languages.

A stream functionality already exists in CORBA with the Externalization service[OMG COSS](see section 2.4.3). The Externalisation service provides a *StreamIO* interface to write and read data to a stream. Components that write or read their data to/from the steam, have to support the *externalize_to_stream* and *internalize_from_stream* operations of the *Streamable* interface. Therefore, getting data in and out of the stream is already well catered for with the Externalization service.

However, the Externalization service does not cater for the setting up a distributed stream and for moving data between the ends of the stream. The Stream Tunnel Service has been devised by extending the Externalization service to supply these mechanisms(Figure 5.1). Consequently, STS is an excellent example of the application of the OMG's Baushaus principle, where a CORBA service is extended to provide the required functionality to serve a particular purpose.

In the stream model, data is written to one end of a stream, moved to the other end of the stream and read out there. How data is moved is an implementation detail of the service, therefore has no effect on how clients use the service. The actual network data transport mechanism used to move data across the distributed environment does not necessarily have to be the ORB. Hence, more efficient network data transport mechanisms could be employed that improve the performance of data transfer. This characteristic of choosing a network data transport is why the service is termed a 'tunnel' as it can go through the defacto communication mechanism i.e. the ORB or can go underneath it by using a mechanism like sockets[Rieken 92] for example.

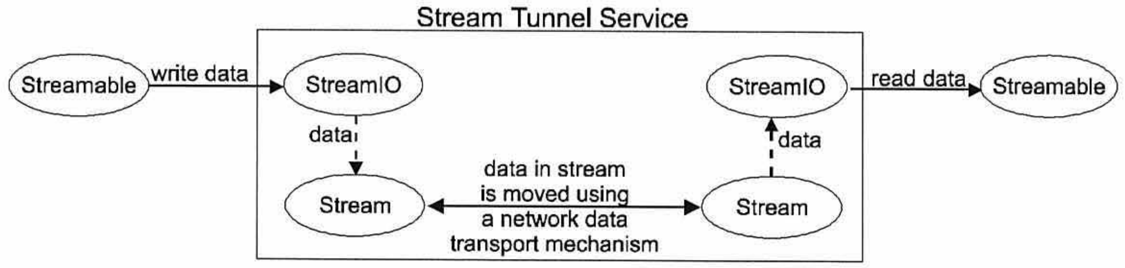


Figure 5.1: *The Stream Tunnel Service extends the Externalization service to provide data transfer between streams*

The contents of the chapter is as follows. Firstly, the interfaces of the Stream Tunnel Service are introduced as well as an explanation on how the components implementing these interfaces interact to provide the distributed stream functionality. Next, further explanations are given explaining the sequence of activities needed to setup a stream and push data through it. The utilisation of streams using low-level data transport mechanisms is presented next. This includes an experiment comparing the performance of streams using

ORB-based and TCP socket-based data transport mechanisms. The implementation of the STS components for the experiment is briefly described. How the experiment was carried out and a discussion of the results of the experiment is also explained. Finally, suggestions are made on other possible applications of the Stream Tunnel Service.

5.2 Interfaces and Components of the Stream Tunnel Service

This section introduces the IDL interfaces of the Stream Tunnel Service(STS) and describes the behaviour of components implementing these interfaces, demonstrating how they work together to carry out the service's purpose. Only segments of the service's IDL code is presented here. The whole STS IDL code can be found in appendix C. This section makes the presumption that the reader is familiar with the interfaces and workings of the Externalization service(see section 3.4.3).

5.2.1 Stream Channels

The most basic component of STS is the StreamChannel. As shown in figure 5.2, the StreamChannel inherits the Stream interface. Using the Stream interface, Streamable objects can be externalised and internalised to/from the StreamChannel. This is the mechanism by which data gets into and out of the StreamChannel. The actual StreamChannel interface adds operations to push and pull data held in the stream to and from the other side of the stream. Therefore, a distributed stream comprises of a pair of StreamChannel components making up the ends of the stream. Data is transported between these two ends with the push and pull operation.

Each StreamChannel component at the ends of a distributed stream has to take on the role of a client or server. A StreamChannel component in the client role drives all

```
interface StreamChannel : CosExternalization::Stream
  {
     void pull_stream_data( in StreamChannelServer sourceChannel )
        raises( NoDataAvailable, StreamNotAvailable, DataTransferError);

     void push_stream_data( in StreamChannelServer targetChannel )
        raises( NoDataAvailable, StreamNotAvailable, DataTransferError);
  };
```

Figure 5.2: *The StreamChannel interface*

```
typedef sequence <octet> StreamedData;

interface StreamChannelServer : StreamChannel
   {
      void push_StreamedData(in StreamedData restoredData)
         raises( CannotAcceptData );
      StreamedData pull_StreamedData()
         raises ( NoDataAvailable, DataTransferError );

      oneway void send_data() raises ( NoDataAvailable, DataTransferError
);
      oneway void receive_data() raises( CannotAcceptData );
   };
```

Figure 5.3: *The StreamChannelServer interface extending the StreamChannel interface*

transport of data between the ends of the stream, while the server side StreamChannel takes on a passive role and is slave to the requests of the client for data. There are two primary reasons for this:- the first reason is to preserve integrity of state of the stream (if both sides were simultaneously trying to move data then problems would occur). The second reason is an implementational issue concerning the CORBA architecture. In CORBA, to receive distributed requests, a component/CORBA object must be running on top of an object adaptor. Consequently, the server side of the stream must also run on top of an object adaptor, for the client side to be able to communicate with it. Generally, client applications operate synchronously and are singularly threaded, hence do not run object adaptors that provide asynchronous communication with the external CORBA environment.

Reflecting the client/server roles of the StreamChannel components, the push and pull operations refer to a StreamChannelServer component as the other side of the stream. The StreamChannelServer interface(Figure 5.3) further extends the StreamChannel interface by adding operations which the client side of the stream can invoke to actually move data between the two sides. The operations of the StreamChannelServer interface are only a basis for possible methods of moving data and their implementation is not obligatory. For more specialised methods of data movement the StreamChannelServer interface should be further extended to provide operations with the required functionality. If a specialised method of data movement is used, then the client side StreamChannel has to accordingly use that method as StreamChannel work in pairs.

The StreamChannelServer interface provides two basic methods of data transfer. The first uses the *push_StreamedData* and *pull_StreamedData* operations to transport the data held in the stream as a sequence of octets i.e. an array of bytes. This method utilises the

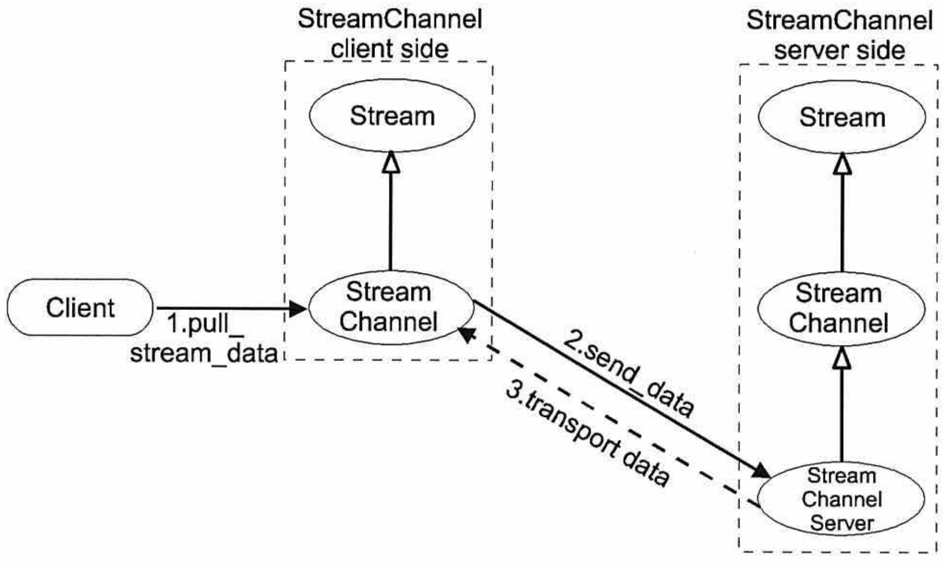


Figure 5.4: *Sequence of requests to transport data from the server to client StreamChannel*

ORB for data transport. The second method allows the use of transport mechanisms other than the ORB. The operations *send_data* and *receive_data* signal the server side of the stream to either send its data or to get ready to receive data. The actual data can then be transported using some mechanism other than the ORB.

Figure 5.4 shows the StreamChannel client and server sides and the hierarchy of interfaces that make the two components. Given that the server side has already had a Streamable object externalised to it, then the figure shows the sequence of requests required to 'pull' the data to the client side using a network data transport mechanism other than the ORB.

5.2.2 Stream Channel Factories

In keeping with the model of creating CORBA objects specified in the LifeCycle service, the StreamChannelFactory interface(Figure 5.5) provides a create operation to return a

```
typedef CosLifeCycle::Criteria TransferChannelCriteria;

interface StreamChannelFactory
  {
    StreamChannel create(inout TransferChannelCriteria
channelType)
       raises( ChannelTypeNotSupported, BaseChannelSupportedOnly,
       ChannelOpenFailed );
  };
```

Figure 5.5: *The StreamChannelFactory interface*

new instance of a StreamChannel. The create operation takes a parameter of TransferChannelCriteria type. This parameter can be used to pass all information concerning the set-up of the stream. For example, it could pass the host address and port number of a socket based.

The TransferChannelCriteria type is based on the LifeCycle's Criteria. This type is made up of a sequence of Named Value Pair(NVP). The NVP type comprises of a structure that contains a string value name and an IDL 'any' type. Therefore, a NVP contains a name describing what the contained value represents. Using a sequence of NVP the information detailing the set-up of the stream can be passed to the factory. This makes stream configuration very flexible as the information detailing the stream set-up can be dynamically constructed. Examples of stream setup information are shown in figure 5.6.

Stream type description	Value string name	Example value
1. Socket based stream	"stream type"	"TCP socket"
	"host address"	"147.345.234.394"
	"port number"	5001
2. ORB based octet block stream	"stream type"	"ORB octet sequence"
	"block size"	1024

Figure 5.6: *Example of stream set-up information contained in a TransferChannelCriteria type*

5.2.3 Stream Tunnels

The StreamTunnel interface makes the use of StreamChannels by encapsulating the stream mechanism and its set-up. It also hides the use of the Externalisation service and Lifecycle service that are necessary for the internalising and externalising ability. Each process using streams will have an instance of a StreamTunnel. The StreamTunnel component will manage all stream connections for the process.

The StreamTunnel interface is the primary component of the Stream Tunnel Service that users of the service interact with to access its functionality. The interface(Figure 5.7) has two sets of operations. One set deals with pushing and pulling Streamable objects through

```
interface StreamTunnel
  {
  StreamChannel push_streamable(
       in CosStream::Streamable streamableSource, in StreamChannel targetStream )
                    raises( NoDataAvailable, StreamNotAvailable, DataTransferError
);

  CosStream::Streamable pull_streamable(
       in CosLifeCycle::FactoryFinder finder, in StreamChannel sourceStream )
             raises( CosLifeCycle::NoFactory, CosStream::StreamDataFormatError,
             NoDataAvailable, StreamNotAvailable, DataTransferError);

   StreamChannel open_channel(
       in StreamTunnelService::StreamTunnel other_tunnelEnd,
       inout TransferChannelCriteria channelType, inout StreamChannel otherEnd )
             raises( ChannelTypeNotSupported, BaseChannelSupportedOnly,
             ChannelOpenFailed );

  void close_channel( in StreamChannel targetChannel )
             raises( StreamNotAvailable );
  };
```

Figure 5.7: *The StreamTunnel interface*

a stream, the other is for creating and closing a stream. The creation of streams is covered in the next section.

The StreamTunnel with its push and pull streamable operations effectively permits any object supporting the Streamable interface to be transported through a stream(Figure 5.8). The prerequisites for transporting Streamables using the tunnel is that a stream must already be setup. Also, for internalisation purposes, each type of Streamable object must have a counterpart Factory object that can be used to recreate the Streamable object. The Factory object must be registered with a Factoryfinder, so that it can be found. For a better understanding of the internalisation process and its use of the LifeCycle service see sections 2.4.2 and 2.4.3.

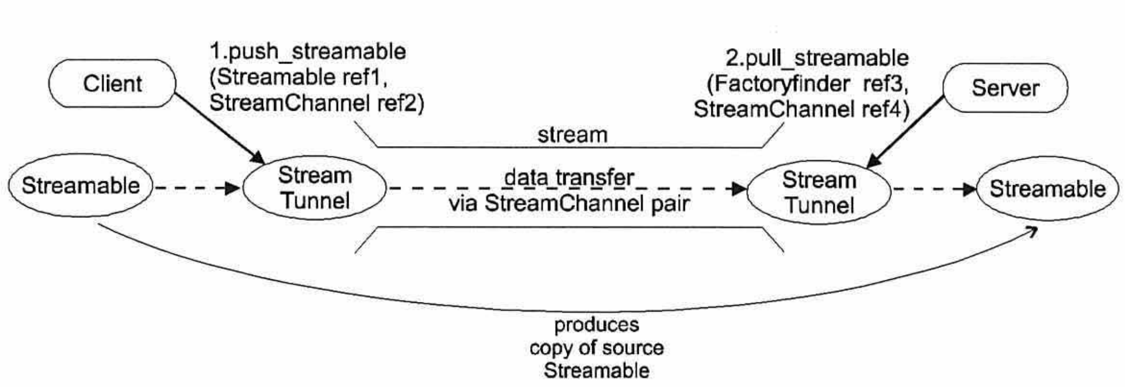


Figure 5.8: *The StreamTunnel can be used to push and pull a Streamable object through a stream*

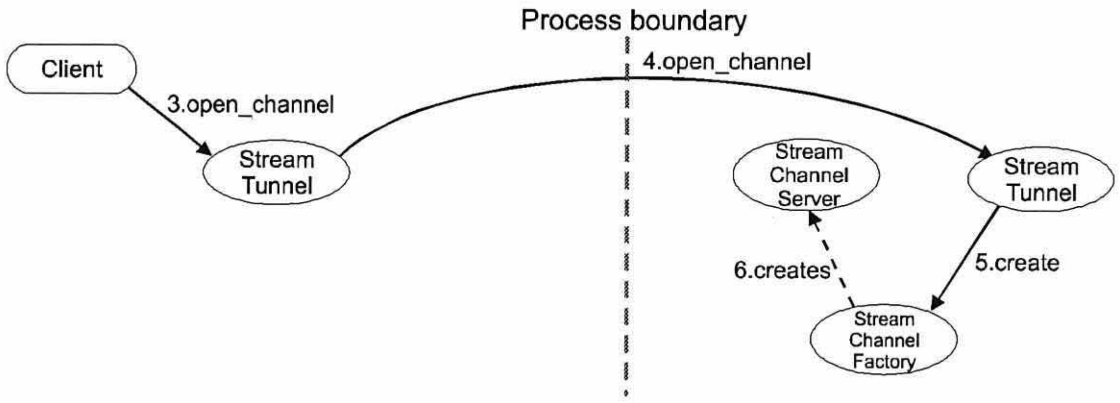


Figure 5.9: *Steps 3-6, requests to create a stream*

5.3 Setting up a Stream

A stream is made up of a client and server side StreamChannel objects. These objects must be created in a synchronised way, so that they are correctly set-up to communicate. The creation of a stream is a complex procedure involving coordinating client and server tunnels and their associated StreamChannel factories. This section describes the sequence of requests and behaviour of these components necessary to set-up a stream. The ORB requests are shown in the accompanying figures.

1. The client of the service must somehow gain references to client and server StreamTunnel components.
2. The client creates a TransferChannelCriteria instance describing the type of stream to be set-up.
3. The client invokes the *open_channel* operation on in its client side StreamTunnel providing the server StreamTunnel and TransferChannelCriteria as parameters.
4. The client StreamTunnel conveys this operation to the server StreamTunnel.
5. On receiving the *open_channel* request, the server StreamTunnel will invoke the create operation on its StreamChannelFactory passing the TransferChannelCriteria parameter.
6. The StreamChannelFactory will examine the information contained in the TransferChannelCriteria and create an instance of a StreamChannelServer that meets the stream type criteria.

7. The TransferChannelCriteria is an IDL 'inout' parameter, which means that any change in the TransferChannelCriteria value is passed back to the caller of the operation. This allows the StreamChannelFactory to add information to the TransferChannelCriteria parameter that will influence how the client side StreamChannel is created. For example, for a socket based stream, the host address and port number could be added.
8. A reference to the newly created StreamChannelServer is returned as a return parameter of the *open_channel* operation to the client side StreamTunnel.
9. The client StreamTunnel calls the create operation of its StreamChannelFactory.

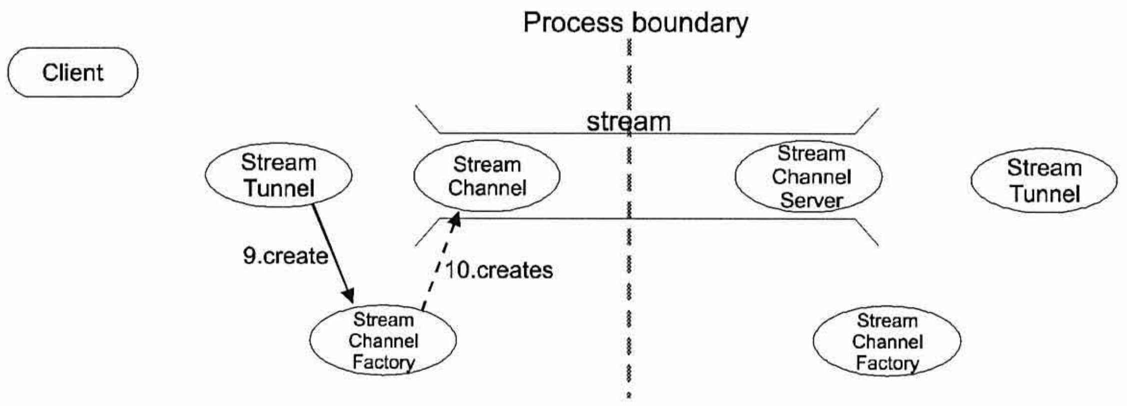


Figure 5.10: *Steps 9-10, requests to create a stream*

10. The StreamChannelFactory creates a client StreamChannel in conformity with the passed TransferChannelCriteria value. Thus, the client StreamChannel is initialised for communication with the corresponding StreamChannelServer. For example, a socket based stream will open a socket with the server at the specified address and port. Consequently, with both StreamChannels in place, the stream is ready for use.

5.4 Transporting Data using a Stream

The behaviour of components and the sequence of requests required to transport data through a stream is presented here for clarity of how the service works. The data to be pushed through the stream is held in a Streamable object that knows how to write its data to/from a stream. Given that a stream is already set-up as described in the previous section, the Streamable object can be pushed through the StreamTunnel using the specified StreamChannel. The sequence of requests are shown in accompanying figures.

1. The client invokes push_streamable on the StreamTunnel, specifying the Streamable and the StreamChannel to push it through.
2. The StreamTunnel is encapsulating the functionality of a client of the Externalisation service, thus it requests the StreamChannel to externalise the Streamable.
3. The StreamChannel requests the Streamable to *externalize_to_stream* passing it a StreamIO interface.
4. The StreamIO interface is used by the Streamable to write its data to the StreamChannel.

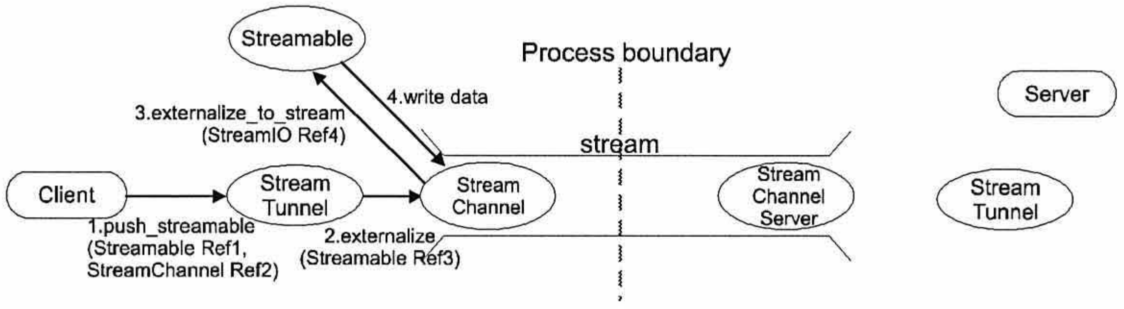

Streamable
Process boundary
4.write data
3.externalize_to_stream (StreamIO Ref4)
stream
Server
Client
Stream Tunnel
Stream Channel
Stream Channel Server
Stream Tunnel
1.push_streamable (Streamable Ref1, StreamChannel Ref2)
2.externalize (Streamable Ref3)

Figure 5.11: *Steps 1-4, requests to externalise an object to a stream*

5. The StreamChannel is now filled with data, so the StreamTunnel requests it to push its data to the other side.
6. The StreamChannels transfer the data using a method that they have implemented and set-up to use.
7. Having transferred the data, the client must now somehow inform the server that the data has arrived and what StreamChannel is holding the data. How this is achieved in outside the scope of STS and is an application level issue.
8. The StreamChannel requests its StreamTunnel to *pull_streamable* specifying the StreamChannel holding the data and a FactoryFinder that can find a Factory object

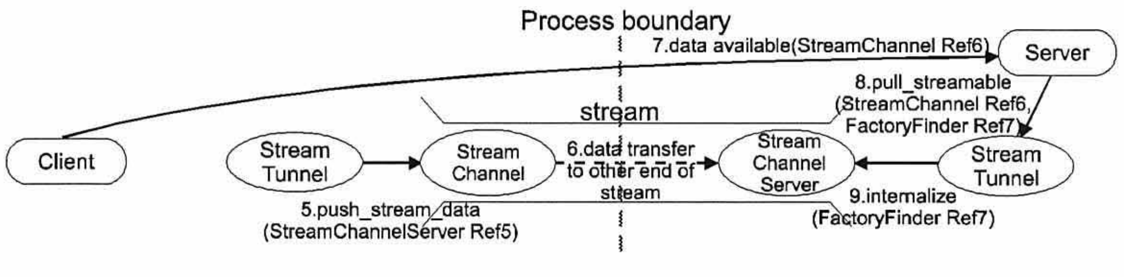

Process boundary
7.data available(StreamChannel Ref6)
Server
8.pull_streamable (StreamChannel Ref6, FactoryFinder Ref7)
stream
Client
Stream Tunnel
Stream Channel
6.data transfer to other end of stream
Stream Channel Server
Stream Tunnel
9.internalize (FactoryFinder Ref7)
5.push_stream_data (StreamChannelServer Ref5)

Figure 5.12: *Steps 5-8, requests to push data through a stream and internalise at the destination*

capable of recreating the Streamable object.

9. The StreamTunnel requests the StreamChannelServer to internalise its data to a Streamable object.
10. The StreamChannelServer needs to find the Factory capable of creating the object that has been externalised to the stream. It does this by extracting information about the object type from the stream, then uses the FactoryFinder object to find the relevant Factory object.
11. The Factory is requested to create a Streamable object.
12. The Factory creates a Streamable object.
13. The StreamChannelServer requests the newly created Streamable object to internalise the data held in the stream.
14. The Streamable object reads its data from the stream.
15. The Server is returned a reference to the newly created Streamable object.

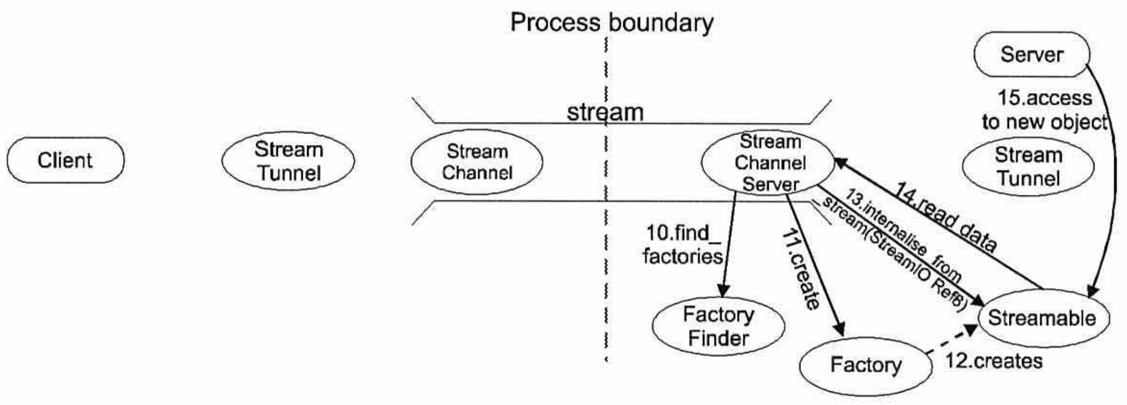


Figure 5.13: *Requests of steps 10-15 to create a Streamable object and internalise the stream's data*

5.5 Streams supporting Low-Level Network Data Transport Mechanisms

STS is providing a wrapper of IDL interface to low-level network data transport mechanisms or ORB based mechanisms. STS allows non-IDL defined data to be transported by these mechanisms and still maintain the integrity of the typed data stored in the stream. This section describes the advantages and disadvantages of using a non-ORB distributed communication mechanism and carries out an experiment to compare streams based on transporting data using the ORB and TCP sockets.

5.5.1 Enhancing Data Transfer using Low-Level Data Transport Mechanisms

The ability to transport data using low-level data transport mechanisms such as TCP sockets, UDP sockets and Transport Level Interface etc [Rieken 92][Stevens 90], can have major performance enhancements for applications requiring fast data transfer. These mechanisms will improve performance over ORB requests as they avoid the following :-

- Any additional communication an ORB performs in a request, such as communicating with ORB daemon processes.
- Complex (de)marshalling of IDL request parameters in stub and skeleton code.
- Bottlenecks caused by object adaptors, as requests have to pass through the object adaptor.
- ORBs based on IIOP/GIOP use connection oriented TCP sockets to communicate. The set-up of these socket connections is dependent on the ORB implementation, thus the manner and therefore the efficiency of how the ORB makes these connections is unknown and out of control of the application. For example, an ORB might make a new connection for each request or route all requests via a central ORB daemon process causing a bottleneck.
- The rate at which data can be transferred is very much dependent on how fast a server can consume data being sent by the client. Usually, servers cache data being received in a buffer area before moving it on to its destination. If an ORB/object adaptor has a small buffer area and a client is sending a large amount of data, then the buffer area might easily be filled. This leads to the halting of data transfer to clear the buffer, thus the data transfer becomes inefficient. A proprietary transfer mechanism that is specific to the transfer needs of a specific application type can specify the size of its own buffer area for maximum efficiency.

For some application types the data transfer rate of ORBs might be unsatisfactory. These applications would benefit from efficient data transfer mechanisms that the stream can support. Such application types are:-

- Applications transferring files.
- Multimedia applications needing to send continuous stream of data.
- Applications transferring enormous multi-dimensional arrays of scientific data such as petroleum industry based applications.
- Applications transferring large collections of objects.

However, using low-level network data transport mechanisms comes with a cost, in that the StreamChannel objects have to provide mechanisms that ensure the data is transferred correctly. The objects have to make sure all data sent is correctly received, so that no data is lost and segments of data sent have to be correctly ordered when received. Also if data is lost then the server has to request the data to be resent. These data transfer consistency issues are usually addressed by the ORB. Another cost with low-level transport mechanisms is that applications become dependent on the transport mechanism and the requirement that the platform the application is implemented on supports the specific transport mechanism. However, transport mechanisms such as sockets are widely available on most platforms, but there might be the possibility of platform specific variations on the use of the sockets. Nevertheless, these costs might be outweighed by the demand for fast data transfer by the certain applications.

5.5.2 Experiment to Compare ORB and Socket Based Stream Mechanisms

An experiment[Ball 98] was carried out to compare the performance of a stream based on transferring the stream's data as sequence of IDL octets(equivalent to bytes) and a stream based on transferring the data as bytes through a connected TCP socket. The experiment will record the time taken to transport increasingly larger sized arrays of bytes. Presented first are the implementation details of the Stream Tunnel Service components needed to carry out the experiment. Next, how the actual experiment was carried out is described and then the results are shown with a discussion analysing the results.

5.5.3 Details and Problems of Implementing the Stream Tunnel Service Components

The ORB utilised to carry out the experiment was the C++ COOL-ORB v4.1 from Chorus Systems. COOL-ORB was not one of the most popular ORBs available and has since been withdrawn when SUN Microsystems took over Chorus Systems. COOL-ORB was chosen as, at the time of starting the project, it was free for trial use and it was the only ORB available for the only platform available for development, which was SUNOS 4 using the GNU C++ compiler. During the progression of the project a Solaris 2.51 platform became available and development shifted over to it, including this experiment. However, the COOL-ORB was kept due to developed code being locked to the proprietary functions of the ORB[Chorus 97] such as ORB initialisation, BOA initialisation, CORBA object registration with the BOA and binding implementation object to skeleton code. This is an example of ORB vendor lock that prevents portability of application code between ORBs. Over the last few years the OMG has standardised these proprietary ORB functions. Consequently ORBs have matured to implement these functions, therefore increasing the portability of code between different ORBs. However, overall developing with the COOL-ORB was very satisfactory as it was reliable and easy to use.

The first component of STS to be implemented was the Streamable object used to hold an array of bytes/octets. This object implements the Externalisation service's *externalize_to_stream* and *internalize_from_stream* operations, as well as set_octet_seq and *get_octet_seq* operations for applications to get data into and out of the object.

The Externalisation service's Stream and StreamIO interfaces were implemented next. These two interfaces are essentially the same component, as the Stream requests Streamables to externalise and internalise data, and also hold the stream's data. The StreamIO interface provides operations to read and write CORBA data types from/to the stream e.g. *write_string* or *read_long*. Data written to the Stream via StreamIO is held in a temporary file as the amount of data written to the Stream is unknown(Figure 5.14).

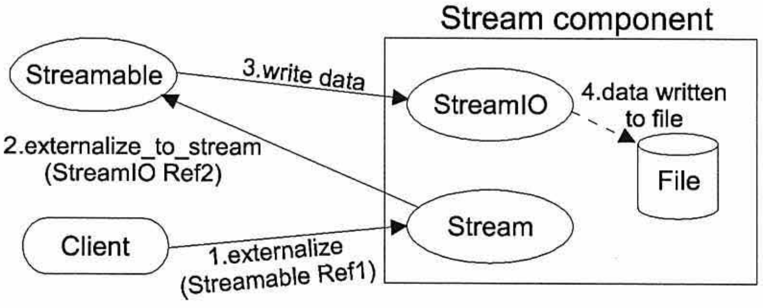


Figure 5.14: *Data externalised to a stream is written to a file*

A problem with the StreamIO interface is that it does not provide for reading and writing sequences(arrays) of data types. Hence, to repeatedly write individual data elements of a large array of elements would be grossly inefficient. To solve this problem the StreamIO is extended to an ExtendedStreamIO interface to provide operations for the reading and writing of sequences of data types e.g. a sequence of octets. Appendix C includes the ExtendStreamIO interface.

The consequence of extending StreamIO so that sequences of data types are written to the stream is that the Externalisation service's Standard Stream Data Format(SSDF) has to also be extended. SSDF specifies the format that data is stored in a stream, including the specification of tag values for each data type that can be written to the stream. Each data type that is written to the stream is stored with an accompanying tag describing the data type. Following this technique, each sequence of data type has to be assigned a tag value(Appendix C). However, this extension of the SSDF is proprietary to this project. Streaming sequences of data types is an essential feature and should not have been left out of the original Externalisation service. A more ideal situation is if the OMG was to specify an equivalent ExtendedStreamIO and SSDF tag values for sequences, thus creating a standard.

The StreamChannel interface extends the Stream by adding push and pull operations to send or get data from the other side of the stream. The StreamChannelServer interface is slave to this at the client side and provides operations to actually initiate the transfer of data. These two interfaces act in pairs that support the same method of data transfer.

The first pair of StreamChannels and simplest to implement for the experiment, transfer streamed data as a sequence of octets using ORB requests. The operations for transferring the octet sequence to and from the server side are provided for in the basic StreamChannelServer interface. The client StreamChannel implementation is simply required to call one of these octet sequence operations on the server. The side holding the streamed data has to take this out of the file, convert it to an octet sequence and put it in

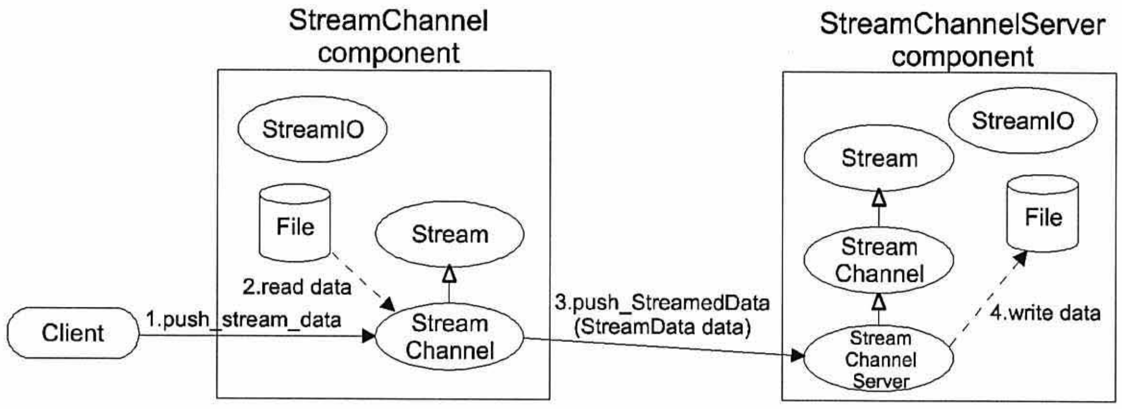


Figure 5.15: *StreamChannels that transfer data using a octet sequence stream*

the returned/passed parameter of the request. The StreamChannel end receiving the data writes it back into a temporary file(Figure 5.15).

The next pair of StreamChannels uses a pair of connection-oriented TCP sockets to transfer data between the two sides. TCP sockets is a low-level network data transport mechanism that provides a reliable data transfer, so that packets of data that are sent are not lost and are received in the correct order. To achieve this reliably, the socket software takes care of packet resending, ordering of received packets and negotiating with sending and receiving sockets to transfer data at a rate that will not overflow the receiving process buffers.

TCP sockets are connection-oriented, meaning that a connection has to be set-up between client and server sockets before any data can be sent. The model for making a connection is of the server socket listening for a connection on a port of its local host. The port the server is listening on is somehow known to the client. The client creates the connection by specifying the host address and the port the receiver is listening. Once the server accepts the connection, either side of the socket can send data through the connection.

The client and server StreamChannels using sockets to transfer data have to follow this model to connect their sockets. The first problem with this is that there is no notion of host address and port number of servers in CORBA communication, as this information is hidden within the object references. Therefore, a mechanism to explicitly let the client know the server address and port to connect to has to be utilised. The STS

TransferChannelCriteria used in passing set-up information between StreamChannelFactorys is the perfect solution to this problem. The flexibility of the TransferChannelCriteria type allows the StreamChannelServer to store its connection information in the type which is forwarded to the client side(see section 5.1.1.2).

The second problem is performance related in respect to the thread behaviour of the StreamChannelServer. This performance problem is caused by server processes being singularly threaded, so that the StreamChannelServer waiting to accept a socket connection results in the whole process being blocked. The time spent waiting for a connection is indefinite, during this time the process is blocked, preventing the object adaptor and any registered CORBA objects from processing further ORB requests. This blocked process may also have extensive implications on the rest of the software system, causing a large degrade in performance.

The solution to this blocking problem is to introduce multi-threading into the server process, so that functions such as accepting socket connections execute on a separate thread to that of the object adaptor and registered CORBA objects. The socket server thread will only block its own thread and not interrupt the whole process.

Some ORBs implicitly support multi-threading in their object adaptors providing different activation policies(see section 2.3.3) such as server-per-method. An activation policy of server-per-thread runs each request received on a separate thread of execution. The multi-threading of requests would avoid this socket blocking problem.

COOL-ORB only supports single threading and a shared server policy. Therefore, to execute the socket connection acceptance on a separate thread, a thread has to be explicitly created using operating system calls. The Solaris operating system offers two kinds of threads, the first is Solaris's own native threads[Sol Man] and the other is POSIX threads[POSIX 97]. Solaris's own native threads permits finer control over the scheduling and execution behaviour of the thread, but is specific to the Solaris platform. POSIX is an international standard for aspects of computing environments including thread creation and execution. The benefit of POSIX threads is that code written using the POSIX thread

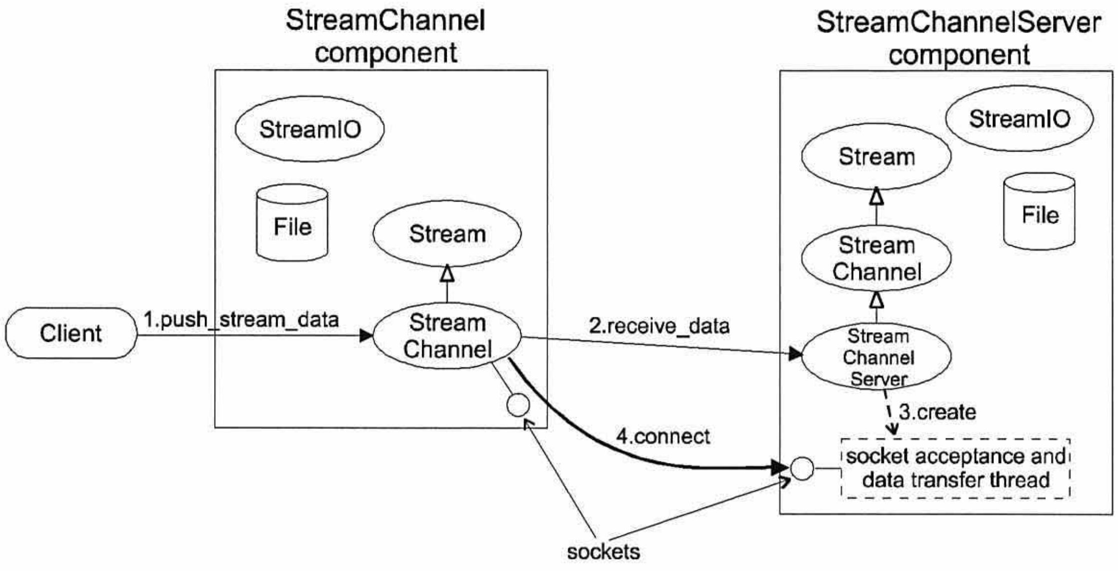


Figure 5.16: *Requests necessary to set-up a socket connection between StreamChannels(steps 1-4)*

functions is portable over any POSIX compliant platform. To prevent further locking of code to specific platforms and software, POSIX threads were utilised to create and run threads for the server to wait for a connection and then to receive or send data.

Figures 5.16 and 5.17 show the sequence of requests and activities to connect a pair of StreamChannel sockets and transfer data across the connection. The steps describing these actions are below. This description makes the presumption that the stream is already set-up and the client StreamChannel has extracted the host address and port number from the TransferChannelCriteria.

1. The client requests the socket StreamChannel to push its streamed data.
2. The StreamChannel sends a *receive_data* request to the StreamChannelServer signalling it to get ready to receive data.
3. The StreamChannelServer creates its socket acceptance thread and starts running it in parallel with the main execution thread of the process. After being started, the acceptance thread will block as it waits for a connection from the client.
4. The client will initiate a connection with the server socket. Upon connection data can be transferred between the sockets.
5. The client StreamChannel reads the data from the temporary file.
6. The data is pushed through the socket connection.
7. The server receiving thread writes the data back to a temporary file.

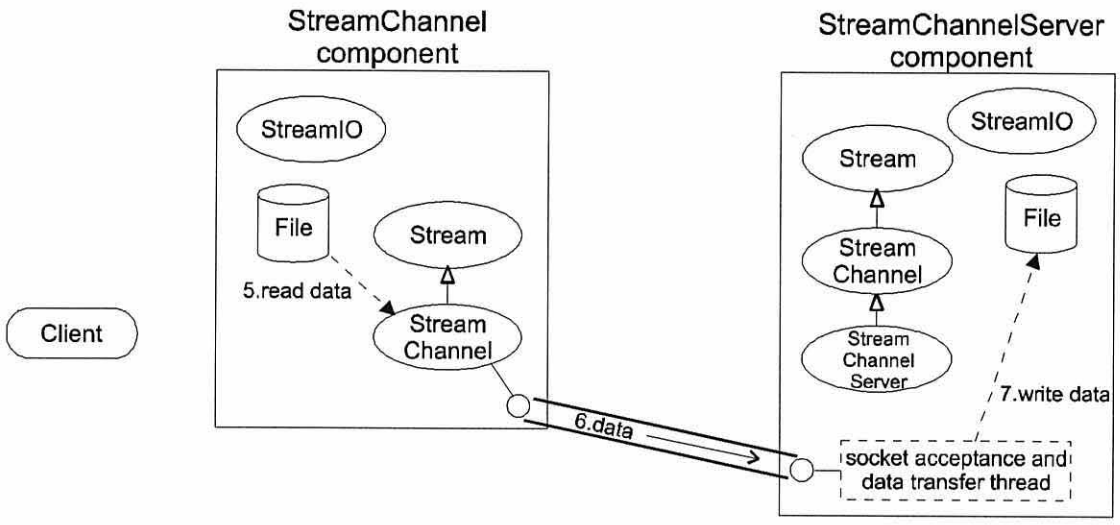


Figure 5.17: *Data transport through the socket(steps 5-7)*

Multi-threading can bring much performance enhancement to systems, especially to servers catering for multiple clients. But it also brings a cost in the complexity of applications. Multi-threaded applications have to organise the creation and scheduling of threads using operating system thread management functions or toolkits that insulate applications from the specifics of platform threads. Another complexity innate in parallel execution is preserving the integrity of shared data between threads. The solution to this classic problem involves keeping different threads from simultaneously manipulating the same data. Thus, concurrency control must be exerted on the shared data allowing a thread to gain exclusive locks on data it wishes to manipulate preventing interference from other threads.

The data being received by the StreamChannelServer must be locked until all the data is received. This is to prevent any other thread accessing the data until it is completely transferred.

Locking of the StreamChannelServer object can be achieved in several ways. ORB products[Orbix][Visibroker] supporting multi-threaded object adaptors generally provide an API that provides an abstraction to the management of native operating systems locks. The API will allow locks to be gained and dropped on instances of CORBA objects. Although this form of locking is independent of specific operating systems, it is dependent upon the particular ORB product. This causes a high dependency between the application and the ORB product. However, this manner of locking cannot even be considered in the implementation of the StreamChannelServer as such functionality is not provided by COOL-ORB.

Another way of locking the StreamChannelServer is for the object to support the Concurrency service's Lockset interface. A client accessing the StreamChannelServer would firstly have to gain a lock on it. The StreamChannelServer will then prevent access from any other than the lock holder. The problem with this solution is that a client would need to gain a lock to make requests on the StreamChannelServer. This would make all components involved unnecessarily complicated, as all that is necessary is some internal mechanism to prevent the methods of a StreamChannelServer being executed while the data transfer thread is running. A solution such as implementing concurrency control via the Lockset interface is more of a system level concurrency solution, where a resource is being accessed by clients. Here, the resource (i.e. the stream's data) is only being accessed by different parts of the same component.

The solution for the StreamChannelServer implementation was for it to have a semaphore[Sol Man] variable. The semaphore acts as a lock to the component's stream data. Each method of the StreamChannelServer object must gain the semaphore before accessing the data. Hence, the data transfer thread has the semaphore while it is executing, any other thread trying to gain the semaphore while this is happening will be blocked.

The downside of using a semaphore solution is that the management of semaphores is specific to the operating system's API, thus causing a dependency. Also, the low-level procedural nature of using operating system APIs is unlike the high-level object-oriented nature of CORBA that the system is based on. This causes a fracture in the internal implementation of the system increasing the complexity and reducing the flexibility of the system.

Ideally, all parts of a component should be implemented from within the same paradigm for simplicity and consistency. Implementing the StreamChannelServer using low-level mechanisms such as threads and semaphores that are from outside the CORBA model results in much complexity being introduced. This complexity comes from merging the two paradigms:- each mechanisms model has to be mapped to the other, with much the same consequences and difficulties as a mapping between two disparate data storage models.

Having the built the StreamChannel ends and the Externalisation service components, next to implement is the StreamTunnel that encapsulates the workings of these components and makes STS easier to use for clients of the service.

The StreamTunnel component's main responsibilities are opening/closing a stream, pushing/pulling data and commanding the StreamChannel to externalise/internalise as described in sections 5.2.3 and 5.2.4. Hence, the implementation of the StreamTunnel is fairly simple as it just automates a sequence of request calls. The only detail not explicitly apparent is that the StreamTunnel must keep a list of streams that are open by recording object references to client and server StreamChannels.

5.5.4 Details of Running the Experiment

The experiment compares data transfer rates of two types of streams. One stream uses a pair of StreamChannels that pass data held in the stream as a sequence of octets in an ORB request. The other stream transfers the data through a connected TCP socket. The data transfer rate is measured by recording the time taken to push increasingly larger sized data chunks through the stream. These data chunks are the state data of Streamable objects that are created with a set size of bytes which can be externalised to or internalised to/from a StreamChannel. The Streamable object is pushed from client to server side through a StreamTunnel that uses a previously opened stream.

Two sets of time recordings are taken. The first set measures the time taken to perform the whole procedure of moving a Streamable from one side to the other. This includes externalising an object on the client side, transporting its data through the stream, signalling the server that the data has arrived and internalising the object on there. The starting time for this procedure is recorded before the client issues the *push_Streamable* operation and the finishing time is recorded on return from the client signalling the server that the data has arrived. The interface and operation that the client uses to signal the server is shown in figure 5.18. The difference between the starting and finishing time is the time taken to transport the Streamable using the StreamTunnel and the specific stream type employed.

```
interface Receiver
  {
    void receive(in StreamTunnelService::StreamChannel sourceChannel);
  };
```

Figure 5.18 *The interface used to signal the server the data has arrived*

The second set of timings measures the time taken to transport the streamed data from one StreamChannel to the other. The starting time is taken before the StreamTunnel issues the *push_stream_data* request on the StreamChannel and the finishing time is taken on return from this operation. This timing for pushing the data across the stream only measures the time taken to read the streamed data out of the file, transport the data between through the stream using one of the transport mechanisms and write it back to a temporary file at the other end. The reading and writing to/from a temporary file has to be performed for each stream type. Therefore, this timing is more representative of the behaviour that using different data transport mechanisms ensues.

The experiment is measuring the time taken to push Streamable objects that hold segments of data. The size of these data segments will vary between 10 kilobytes and 12 megabytes. The steps between these limits is calculated using the formula $(n^2)*10000$ bytes, where $n = 1$ to 35 in steps of 1. For each stream type, six runs will be made of sending these ranges of data segment sizes through the stream and recording their time. The runs are numbered 1 to 6. Odd runs will be made with the data segment sizes increasing and even runs with the data segment sizes decreasing. This is to counter any memory allocation problems with reserving such large areas memory.

The experiment was performed on two SUN Sparc 5 workstations running the Solaris 2.51 operating systems. The workstations are connected to the local area network that is made up of a 10Mbps ethernet. The experiment was run at a time of day when network activity was negligible and other activities executing on the machines was insignificant. As already stated, the ORB used was the Chorus COOL-ORB release 4 with the system components implemented in C++ using the GNU C++ compiler. The efficiency of getting data into and out of the stream is increased by providing the ExtendedStreamIO interface, that allows the data to be written/read to/from the stream as a sequence of octets.

5.5.5 Experiment Results

The experiment results are discussed here. A graph showing the experiment results are shown in figure 5.19. The discussion includes the performance behaviour of the streams, explanations are offered to explain this performance behaviour and conclusions on use of different data transport mechanisms for streams.

The graph shows average time in seconds against number of bytes of the size of the data segment pushed through a stream. The average time is calculated from the timings of the six runs carried out for each stream. The black lines show the average timings for the ORB octet sequence based stream. The grey lines show the average timings for the TCP socket based stream. Each stream has two sets of timings that are also shown on the graph. The solid line is the average time for *push_Streamable* operation, therefore the time for the

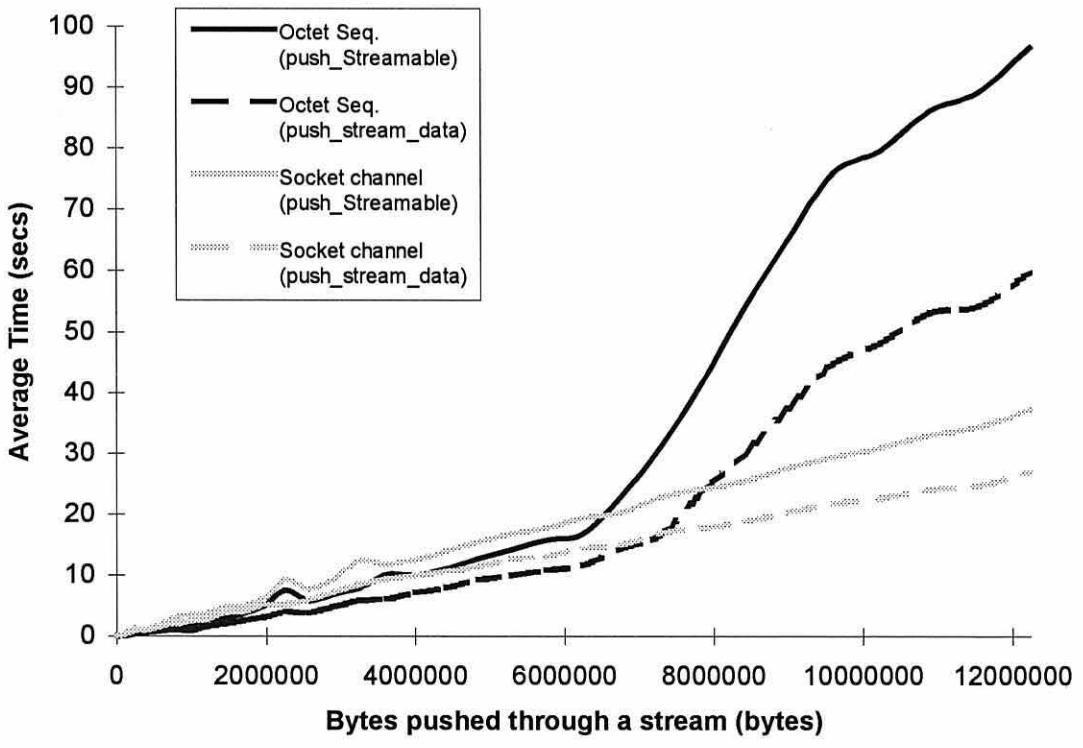


Figure 5.19: *Number of bytes pushed through a stream against time taken*

whole procedure of externalising, data transport and internalising. The broken line is the average time for the *push_stream_data* operation, thus encompassing the reading data out of a file, data transport across the stream and writing back to a file.

The first noticeable feature of the behaviour of the timing lines is that for each stream, the shape of the *push_Streamable* line follows the shape of the *push_stream_data* line i.e. the solid line has a similar shape to the broken line. This proves that any differences in timing behaviour between the stream types is only dependent on the different data transport mechanisms and the externalisation/internalisation actions are linearly proportional to the size of the data segment.

The most prominent and significant feature of the graph is the sudden increase in gradient of the ORB octet sequence stream lines at 6 megabytes. After 6 megabytes the octet stream line climbs steeply, but the socket stream stays at a constant gradient that is proportional to the data segment size. At 9 megabytes and above the octet stream levels out slightly, but at this stage this stream is approximately three times slower than the socket stream. This is evidence that the socket-based stream is more efficient than the ORB-based stream for transporting very large amounts of data.

A possible explanation for this behaviour of the ORB octet sequence stream is related to the manner with which each stream assembles data for transferring using its data transport mechanism. The octet sequence stream has to read all its data out of the file into a memory area before presenting it to the ORB as a parameter in a request. The problem with doing this with very large segments of data is that the data becomes larger than physical memory can handle. When this happens, virtual memory starts being used. This results in disk accesses being necessary to store the data, therefore slowing down the transfer of data by the ORB.

The socket stream assembles data in a different fashion. It reads data out of the temporary file in small chunks of 16 kilobytes. When a chunk of data is sent, the stream fetches another chunk and repeats this routine until all the data is sent. The benefit of this approach is that a large of amount of memory that might require slow virtual memory access does not need be reserved.

Another possible explanation for the inefficiency of the ORB octet sequence stream could be due to inefficiencies of the ORB server receiving the actual data. For example, the size of the data being sent might be larger than its data buffers, resulting in an overflow and the stalling of data transfer. Another source of inefficiency might be the skeleton code and its handling of received data.

Figure 5.20 shows a clearer picture of the graph for data segment sizes of 8 megabytes and under, only for the time taken for the *push_stream_data* operations to perform. The graph plainly shows that the octet sequence stream is 1.5 to 2 seconds faster than the socket stream for the range 1 to 7 megabytes. Beyond this range the octet stream experiences its steep climb. Thus, the ORB octet sequence stream is faster for smaller data segments.

A possible cause of this roughly constant lagging of the socket stream is due to the fact that

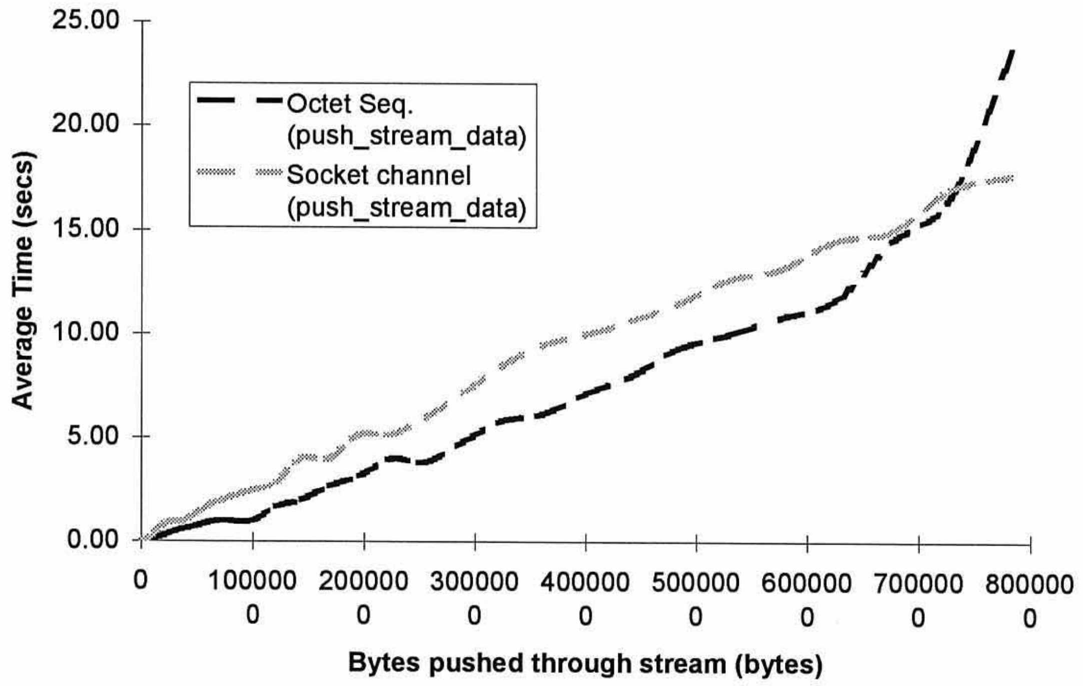


Figure 5.20: *A closer look at time taken to transport data segments under 8 megabytes*

each time it wants to send data it has to remake its connection with the server side socket. Therefore the time lag is due to the StreamChannel issuing the *receive_stream_data* on the StreamChannelServer, the StreamChannelServer creating a thread to accept a socket connection and transfer data. Meanwhile the client StreamChannel is trying to set-up a connection with the server socket. This lag is inherent in the design of the socket stream, but as the size of the data increases, the proportion of time required to set-up the connection compared to the amount of time to transfer the data becomes increasingly insignificant.

This experiment has proven that wrapping low-level network data transport mechanisms in CORBA wrappers has performance benefits over simply using ORB requests for data transport. Although the overhead in setting up a connection outweighs its use for small to medium sized amounts of data, for large amounts of data its use is very valuable. Typical applications that would benefit from such efficient data transfer mechanisms are Exploration and Production industry applications and multimedia applications.

5.6 Further Applications of the Stream Tunnel Service

The Stream Tunnel Service(STS) is very useful in fulfilling the task it was designed for; that is, distributed transport of non-IDL defined data using streams. It has also been proven as a mechanism to improve the performance of the transport of large amounts of data against standard ORB request data passing. However, STS proves to have many more useful functions than was firstly perceived. This section presents other applications where STS could be of use.

STS could be used as the transport system of a persistence system, so that the persistent state of a CORBA object could be indicated and transported through STS to the CORBA object. A key feature of supporting persistence with STS is that all an object has to do to have its state initialised by STS is to support the *externalize_to_stream* and *internalize_from_stream* operations. This allows CORBA objects to read and write their data to/from the stream. This is assuming that there is some way to indicate the location of the persistent state and a means of setting up a server side to the stream.

In regard to the OpenSpirit architecture of persistence(Section 3.2.2), STS could be of used to transport persistent state data to OpenSpirit objects. This would remove the need for the use of the low-level JNI layer and the need for the definition of wrapper objects for OpenSpirit object types. Instead the OpenSpirit objects will simply implement the Streamable operations. This would allow the OpenSpirit objects to be serialised to and from a stream allowing the transportation of their state data. OpenSpirit objects can also benefit from the performance enhancements of using low-level network data transport mechanisms. As OpenSpirit is based in the exploration and production industry, the majority of data will be enormous collections of data resulting in performance problems in transporting.

The performance enhancements and flexibility in the actual method of data transport is also of great relevance to multimedia applications. Typically, multimedia requires the transport of large files or the transmission of a constant stream of data. STS can be used to wrap efficient mechanisms supporting these kind of needs. For example, User Datagram Protocol(UDP) sockets could be used to transmit data very quickly. A UDP socket is connectionless, meaning no connection is made between a socket pair. Packets of data are simply sent from the sender socket with an attached receiver address and port. The arrival of the packets of data at the receiver is not guaranteed, nor is the order in which the packets arrive in. Protocols to guarantee consistent delivery of data has to be explicitly implemented by the applications that are communicating. Consistent delivery will involve application requesting retransmission of data packets, ordering of arriving data packets and an agreement to the rate that data packets are sent as to not flood the receivers buffers with data.

For multimedia applications the benefits of the high speed of UDP socket data transmission would be substantial, especially for constant streaming of data. The consistent delivery of data packets of multimedia information is not always an essential factor. The occasional data packet that is lost might not make much difference to the playing of some video, or the occasional data packet received out of order could merely be discarded. UDP sockets enable unreliable fast transmission of data, but a StreamChannel pair implementing wrapping to such a mechanism still has the benefits of reliable ORB communication. An example where this reliable communication would be necessary is in negotiating the data transfer rate between the StreamChannel ends. At the set-up of a stream, the client side could send test data through the socket to the server end. The client could then query the server using reliable requests for the optimum data transfer rate to use. They could even work out the optimum route to send data packets through the network. Techniques like this and the use of IDL wrappers for fast network data transmission mechanisms make the use of multimedia a reality in CORBA.

In CORBA, objects can only be passed by reference, meaning only object references to CORBA objects can be passed in requests. Actual CORBA objects cannot be passed, thus there is no pass-by-value. If an CORBA object needs to move between process spaces, then some proprietary method must be employed to create the same CORBA object type in the destination process. The state data of the source object then has to be copied to the newly created object. STS innately provides this copy-by-value functionality with the prerequisites that the relevant object factories are available and there are StreamTunnels active in the source and destination processes.

A service that would also benefit from the functionality of using low-level network data transport mechanisms is the OMG's Event service. The Event service allows a form of de-coupled communication between components, in that client components will not necessarily know the components receiving their communication. The model of this de-coupled communication makes use of events, where an event is a signal that some occurrence has happened and can optionally contain data specifying the type of occurrence. Components producing events are event suppliers and send their events to event channels. Components interested in the occurrence of these events are called event consumers. The event consumer registers with the event channel that they are interested in the events of, so that any events sent to the event channel are forwarded to the event consumer(Figure 5.21).

There can be many suppliers and many consumers of events for a single event channel. The Event service is normally implemented using standard ORB requests. Hence on arrival of an event at a channel, the channel has to issue an ORB request to each individual event consumer to signal the occurrence of the event. This would very time consuming for a large

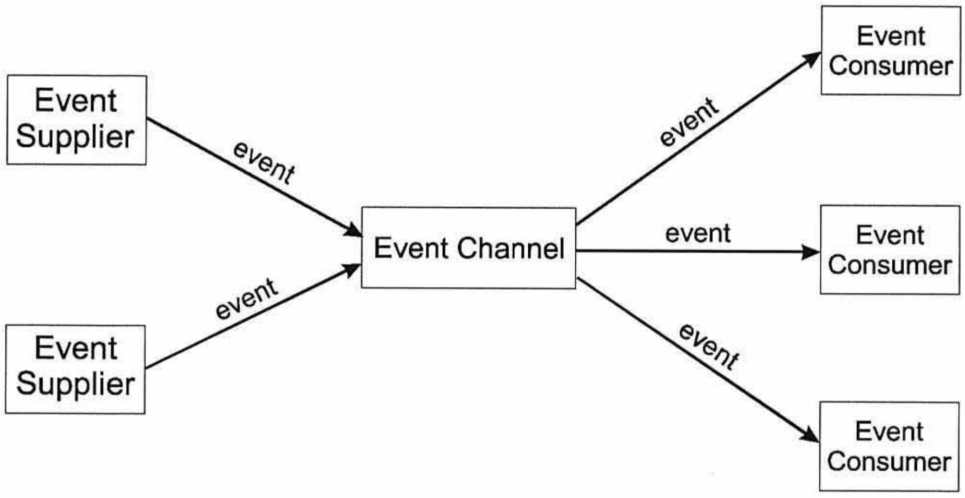


Figure 5.21: *Supplier send events to a event channel and consumers receive these events*

number of event consumers. A more efficient model would be for the event channel to send a single signal and all event consumers to receive it. This functionality is not currently provided for within ORBs, but a mechanism that does provide it is UDP socket broadcasting.

Broadcasting is the network facility whereby packets of data sent out into the network are received and processed by every node(computer) on the local network. If any nodes have processes bound to the port that the data packet arrives at, then it is forwarded to the process for further processing. If no processes are bound to the port then the data packet is discarded.

The flexibility of STS allows it to encapsulate UDP broadcasting permitting a single client end of a stream to have multiple server ends. Data pushed through the client end is transported to each server end using efficient broadcasting, rather than making individual connections with each server end. STS using broadcast streams could be used as an efficient means of transporting events to multiple event consumers(Figure 5.22). Although this model is not in strict keeping with the Event service that uses direct ORB requests to communicate, it is only an optimisation for situations where ORB requests would be too inefficient. Also, STS usage would be totally hidden from client applications by Event service interfaces, therefore normal and STS-based Event service implementations could be interchanged without changing any application code.

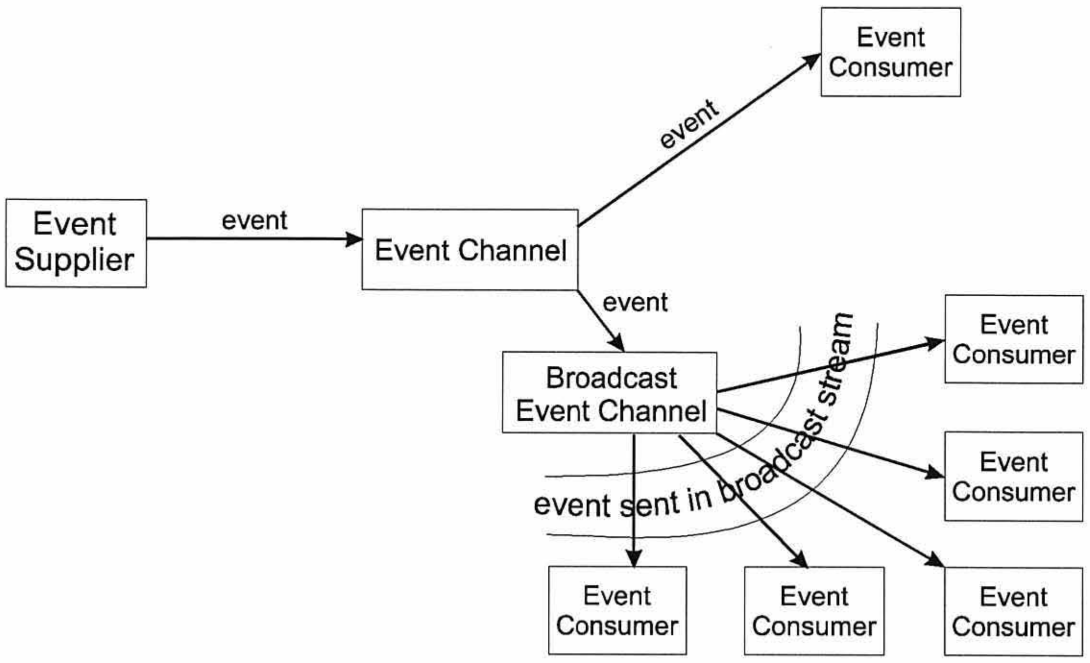


Figure 5.22: *Events are broadcast simultaneously using the broadcast stream*

Another application where STS could be of use is in security and the encryption of data that is sent across the network. An application wishing to secure its data has to explicitly encrypt its data before sending it in an ORB request. Using STS, a StreamChannel pair could be implemented that implicitly encrypts any data sent between them. The StreamChannel pair could also negotiate encryption algorithms and encryption keys. This takes the burden of encryption from the application.

5.7 Summary

This chapter has presented the Stream Tunnel Service and its use in transporting non-IDL defined data. The service extends the OMG Externalization service to provide a distributed stream. This is achieved by pairing StreamChannel components together and providing communication mechanisms to pass data between them. The StreamTunnel component encapsulates the use of the Externalization service and hides the complexity of using and setting up streams.

The StreamChannel components encapsulate the network data transport mechanism that actually transfers data, therefore the method of transferring data can be mechanisms other than the ORB. For example, low level mechanisms such as connection TCP sockets or connectionless UDP sockets could be used for data transfer. These mechanisms would provide performance gains over using the ORB for data transfer. An experiment was carried out to compare transferring data through a stream as an octet sequence of an ORB request, against sending data through a connected TCP socket. The experiment compared sending increasingly larger sized blocks of data through a stream using the ORB and a stream using a TCP socket. The experiment showed that for blocks of data under 6 megabytes the ORB-based stream was faster due to the extra time the socket stream needed to make a connection. Above 6 megabytes of data the socket stream proved more efficient, as the performance of the ORB stream takes a steep decline.

The Stream Tunnel Service has many other uses such as:-

- Transport mechanism for CORBA object state data, enabling migration of objects.
- A persistence mechanism to transport data between a datastore and a CORBA object, given some way to identify and manage the persistent data.
- Use in multimedia applications for constant streaming of data. The data transport mechanism can be optimised to application needs such as use of connectionless datagram based protocols, refinement of buffering routines, negotiation of data transfer rates, packet routing opitmisation and refinement of data consistency protocols.

- Use of broadcasting protocols to provide an efficient Event service with many event consumers.

There is much overlap in functionality between the Stream Tunnel Stream and the ORB. both are data transport mechanisms and retain the types of passed data items. STS is not a replacement for the ORB, but a supplement to it. STS offers additional facilities that the ORB does not, especially the facility to transport of non-IDL defined data and the utilisation of data transport mechanisms with desired performance and reliability characteristics.

In relation to providing a Persistent Data Access service, STS meets the design criteria for a mechanism to transport data objects across the distributed environment. Therefore, STS will be reused by other services as transport mechanism to move data.

Chapter 6

The Data Object Service

6.1 Introduction

The Data Object Service(DObS)[Ball 98] is a set of generic interfaces that permit the management of data objects, where a data object is any type of persistent data that is stored in some kind of datastore e.g. files, object-oriented or relational databases. Management of these data objects by DObS includes the identification, creation/deletion and retrieval/storage of data objects. DObS does not specify operations for the actual manipulation and identification of data objects, as this is left to interfaces that extend the DObS interfaces. Thus, DObS is only an abstract set of interfaces that enforce how the basic data object management tasks should be carried out. One aspect of data object management that DObS enforces is the mechanism that is used to transport data objects between clients and datastores (that is the Stream Tunnel Service).

The Stream Tunnel Service(STS) is a mechanism for transporting non-IDL defined data. The Data Object Service(DObS) provides a mechanism for the management of data objects that are also not defined in IDL. DObS compliments STS by providing facilities to identify what data is to be moved across a stream. Together they provide a simplistic data access and transport system, which is why they are used for an internal mechanism for the transport of data objects between client and server sides of the Persistent Data Access Service.

The Data Object Service(DObS) will prove useful in the management of simplistic data objects such as files or simplistic objects. To provide this functionality, the DObS interface has to be extended to provide operations that will enable the identification and manipulation of specific data object types. The File Data Object Service(FileDObS) is an example of interfaces extending DObS to provide access to and manipulation of files. File DObS provides a distributed file system functionality to CORBA applications.

There are many similarities between the architectures of DObS and the Persistence Object Service(POS)(see section 3.3.1). However, the design of DObS was influenced by the major shortcomings of POS, specifically the lack of a common data transport mechanism hence the Stream Tunnel Service.

Included in this section covering the Data Object Service is the following. Firstly the interfaces of DObS are explained along with components implementing these interfaces interact. Next, the File Data Object Service is presented, explaining its extensions to DObS interfaces that allows it to provide distributed access to files. The actual implementation of File DObS is discussed, including an application that makes use of FileDObS called CORBAFileView. CORBAFileView is a Java application that allows the viewing and manipulation of files. The functions CORBAFileView provides are the viewing of images, text editing, directory manipulation and an example of how File DObS can be used as a simplistic persistence service with object states stored in files. The differences and similarities between the Data Object Service and the Persistent Object Service are discussed. The chapter concludes by explaining some important lessons that have been learnt from developing these CORBA applications and services. These lessons are relevant to CORBA development as well as to general software development.

6.2 The Interfaces of the Data Object Service

This section describes the Data Object Service(DObS) interfaces/components and how they interact with each other to carry out the service's purpose. An overview of the DObS components are shown in figure 6.1. Briefly, the components have the following roles :-

- DOb_ID - identifies the location of data objects to be manipulated.
- Data Object Manager - manages data objects in the client and provides operations to manipulate data objects. It also requests the Data Object Server to retrieve and store data objects.
- Data Object Server - stores and retrieves data objects in its supported datastore.
- Stream Tunnel - used to set-up streama for passing data objects between manager and server.

Interfaces are shown separately here. The complete IDL code for DObS can be found in appendix D.

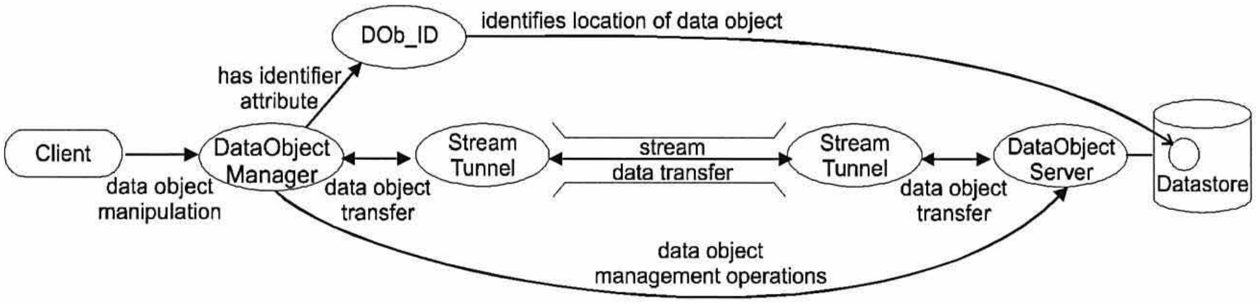


Figure 6.1: *Components of the Data Object Service*

6.2.1 The Data Object Identifier Interface

The Data Object Identifier(DOb_ID) interface(Figure 6.2) is the parent of all interfaces that will be used to indicate the location of data objects. DOb_ID does not contain any actual values to specify the location of the data object within the datastore. It is the responsibility of the extension interfaces of the DOb_ID to add attributes to specify this. For example, an extension of DOb_ID indicating the location of a file would add attributes to indicate the directory and file names of the file. Another example would be to add a integer value representing an object identifier to a DOb_ID to indicate the location of an object in an object store.

The DOb_ID interface does provide an attribute to reference the server that is wrapping the datastore containing the data object. It also provides the operations to translate the information held in the data object identifier to and from a string form. The format of this string is specific to the DOb_ID extension type. The benefit of providing these conversion operations is that the information identifying the data object can be passed by value in the form of a string, instead of passing it as an object reference. If a DOb_ID object reference was passed, then remote requests would have to used to query the information contained in the DOb_ID, which would be highly inefficient.

```
typedef string DOb_ID_String;

interface DOb_ID : CosLifeCycle::LifeCycleObject
  {
    attribute DataObjectServer server;
    DOb_ID_String get_stringified_DOb_ID( );
    void set_DOb_ID( in DOb_ID_String DOb_string_identifier )
      raises (DOb_ID_Invalid );
  };
```

Figure 6.2: *The Data ObjectIdentifier interface*

6.2.2 The Data Object Server Interface

The DataObjectServer interface(Figure 6.3) provides a wrapper to datastores storing data objects. The interface operations provide basic data object management operations and are

```
interface DataObjectServer
  {
    StreamTunnelService::StreamTunnel get_StreamTunnel( );

    void create( in DOb_ID_String DOb_identifier )
       raises(DOb_ID_Invalid, DOb_CreateDenied);

    void retrieve( in DOb_ID_String DOb_identifier,
                   in StreamTunnelService::StreamChannel transfer_channel)
      raises( DOb_AccessDenied, DOb_ID_NotFound,
              StreamTunnelService::StreamNotAvailable );

    void store( in DOb_ID_String DOb_identifier,
                in StreamTunnelService::StreamChannel transfer_channel)
      raises( DOb_UpdateDenied, DOb_ID_NotFound,
              StreamTunnelService::StreamNotAvailable );

    void remove( in DOb_ID_String DOb_identifier )
      raises( DOb_RemovalDenied, DOb_ID_NotFound );
  };
```

Figure 6.3: *The Data Object Server interface*

invoked by the DataObjectManager. Components implementing the interface have the duty of retrieving/storing data objects to/from the datastore and passing the data through the stream to the DataObjectManager.

The operations of the DataObjectServer interface give access to the StreamTunnel, allowing the manager to create streams with the server. Each of the other data object management operations has a string form of the data object identifier parameter indicating the target data object. The retrieve/store operations have an additional StreamChannel parameter indicating the stream to pass the data through.

6.2.3 The Data Object Manager Interface

The DataObjectManager interface(Figure 6.4) is the base interface of all components that manipulate data objects. It provides the basic management operations for data objects. Actual manipulation of data should be provided for in extensions of this interface. Requests of these basic management operations are conveyed to the DataObjectServer. The data object that these operations are performed on is indicated by the target_DataObject identifier attribute. When a management operation is invoked, the identifier attribute's stringified form is obtained and passed in the request to the DataObjectServer. The final operaton - remove_manager simply deletes the manage when it is no longer of use.

```
interface DataObjectManager
{
  attribute DOb_ID target_DataObject;

  void create( ) raises(DOb_ID_Invalid, DOb_CreateDenied);

  void retrieve( )
    raises( DOb_AccessDenied, DOb_ID_NotFound, DOb_ID_Invalid,
          StreamTunnelService::StreamNotAvailable );

  void store( )
    raises( DOb_UpdateDenied, DOb_ID_NotFound, DOb_ID_Invalid,
          StreamTunnelService::StreamNotAvailable );

  void remove( )
    raises( DOb_RemovalDenied, DOb_ID_NotFound, DOb_ID_Invalid
);

  void remove_manager( );
};
```

Figure 6.4: *The Data Object Manager interface*

6.2.4 The DataObjectManagerFactory Interface

The DataObjectManagerFactory interface(Figure 6.5) provides a *create* operation to create DataObjectManager components. There can be many types of DataObjectManager. The type that is needed is dependent upon the type of data object to be manipulated. The type of DataObjectManager to create is specified by the Key attribute type.

```
interface DataObjectManagerFactory
{
  DataObjectManager create( in CosLifeCycle::Key
manager_interface_type,
                           in DOb_ID initial_DOb_identifier)
    raises (DOb_ID_Invalid,NoInterfaceMatchingKey);
};
```

Figure 6.5: *The Data Object Manger Factory interface*

6.3 Using the Data Object Service

This section describes the steps involved to identify, retrieve, manipulate and store a data object. This will give a clearer idea on how the DObS interfaces are used and how the components interact with each other.

1. A client somehow obtains a reference to a DataObjectServer containing the data objects it wishes to manipulate.
2. The client creates a Data Object Identifier(DOb_ID). The client sets the target server attribute and the attributes indicating the location of the data object.
3. The client requests the DataObjectManagerFactory to create the relevant DataObjectManager.
4. The factory creates the relevant DataObjectManager.

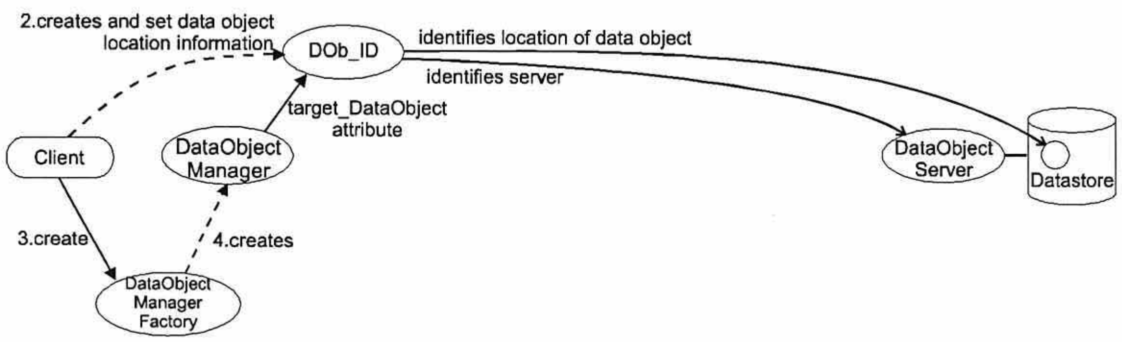

2.creates and set data object location information
DOb_ID
identifies location of data object
identifies server
target_DataObject attribute
Client
DataObject Manager
DataObject Server
Datastore
3.create
4.creates
DataObject Manager Factory

Figure 6.6: *Steps 1-4 of using the Data Object Service*

5. The DataObjectManager component needs to setup a stream with the DataObjectServer. Therefore, it uses the *get_StreamTunnel* operation to acquire the server side StreamTunnel reference.
6. The DataObjectManager uses the server's StreamTunnel to setup the stream.
7. The client invokes the retrieve operation on the DataObjectManager.

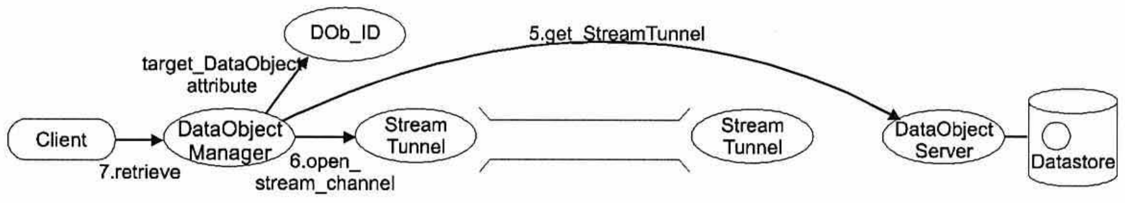


Figure 6.7: *Steps 5-7of using the Data Object Service*

8. The DataObjectManager gets the stringified version of the DOb_ID. It invokes the retrieve operation on the DataObjectServer passing the stringified DOb_ID and the server end StreamChannel.
9. The DataObjectServer recreates the DOb_ID object using the information the passed DOb_ID string.
10. The DataObjectServer examines the information in the DOb_ID and retrieves the indicated data object from the datastore.

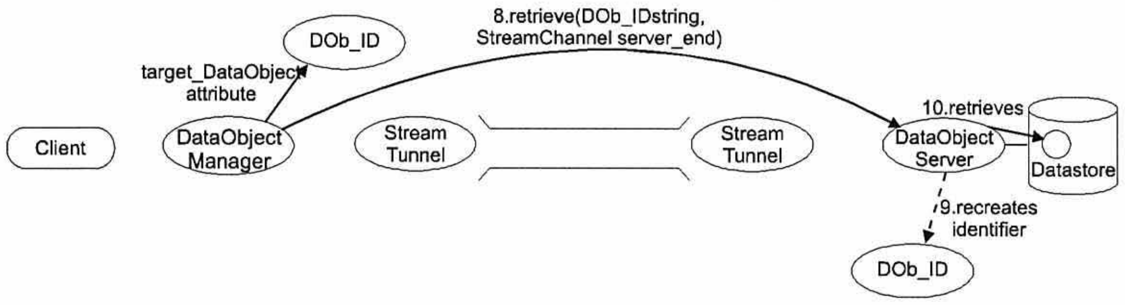


Figure 6.8: *Steps 8-10 of using the Data Object Service*

11. The DataObjectServer produces the retrieved data object in the form of a Streamable object.
12. The DataObjectServer pushes the Streamable into the StreamTunnel.
13. Control is passed back to the DataObjectManager that pulls the data object through the stream.

14. The client manipulates the retrieved data object using interfaces that extend the DataObjectManager interface.
15. Having finished manipulating the data object, the client will invoke the store operation.
16. The data object is stored by carrying out the same process as to retrieve the data object, except the store operation is used and the data object travels the opposite direction through the stream.

6.4 The File Data Object Service

The Data Object Service(DObS) specifies basic data object management operations and how components interact to get data objects into and out of a datastore. However, DObS is only abstract; it does not provide identification and manipulation details for specific types of data objects. These details are provided for in extensions of the DObS interfaces. The File Data Object Service(FileDObS) is one such extension of DObS, providing the detailed operations for identification and manipulation of data objects that are files stored in file systems.

The interfaces of the File Data Object Service are described here. Inherent to the access of files is the ability to navigate and manage directories, so that file listings of directories can be obtained and directories can be created/deleted. To provide this ability, FileDObS contains two sets of extensions to the DObS. One set deals with the management of file data objects, where the data object contains simple data. The other set views directories as data objects, where the data object provides information on the contents of a directory.

A brief description of the implementation of the FileDObS is given. This implementation takes the form of a Java language client side and a C++ COOL-ORB server side. Java was chosen due to its exceptional portability features preventing locking to a particular platform. The co-operation between the two sides of the service demonstrates CORBA's excellent ability of allowing inter-operation between differing platforms and languages.

The Java implementation of the client side of the FileDObS provides a type of distributed network file system to Java applications. This is especially important to Java applets that are embedded within web pages and execute within the confines of an internet browser. The internet browser enforces strict security that prevents the applet accessing local file systems. Using FileDObS, the applet would be able to access remote file systems instead.

CORBAFileView is Java application demonstrating the use of the client FileDObS implementation. CORBAFileView allows the graphical navigation of directories and the selection of files to retrieve. Functions that CORBAFileView allows on files include the viewing of image, the retrieval, editing and storage of text files and a limited demonstration of how FileDObS can used to provide persistence to CORBA objects.

6.4.1 The File Data Object Service Interface Extensions

The File Data Object Service(FileDObS) extends the generic Data Object Service to provide interfaces containing operations that specify how data objects representing files can be identified and manipulated. FileDObS also describes interfaces that allow directories to be represented as a data object, where the data object contains information on the file contents of the directory. Presented first are the set of interfaces providing access to directory information.
The DirectoryDOb_ID interface(Figure 6.9) provides identifier information on the location of a directory, therefore the fpath attribute is set to the path name of the directory.

```
interface DirectoryDOb_ID : DataObjectService::DOb_ID
  {
    attribute string fpath;
  };
```

Figure 6.9: *The Directory Data Object Identifier*

The DirectoryDObManager interface(Figure 6.10) is the manager interface for the directory

```
enum ftype{ FILE, DIRECTORY };
struct Content
{
  string fname;
  ftype file_type;
};
typedef sequence <Content> Contents;

interface DirectoryDObManager:DataObjectService::DataObjectManager
  {
    Contents get_contents() raises(NoDirectoryDataAvailable);
  };
```

Figure 6.10: *The Directory Data Object Manager interface*

```
//DirContents for private use of DirectoryDObManager

interface DirContents : CosStream::Streamable,
                        CosLifeCycle::LifeCycleObject
  {
    attribute Contents fileList;
  };
```

Figure 6.11: *The Directory Contents interface*

data object. It provides an operation to get the contents of a directory, where the contents describe contained file and directory names.

Data objects are transported between the manager and the server using a stream. Data that is pushed through a stream is held in a Streamable object, therefore each data object type must specify a Streamable object to hold its data. The DirContents interface(Figure 6.11) is an extension of the Streamable interface providing a Streamable component holding directory information.

The interfaces just presented allow the fetching of directory information. The interfaces include operations to enable the identification of directories, retrieval of directory information and the transportation of the information through streams. The interfaces for the manipulation of file data objects take a similar form in providing extensions of the DObS interfaces, but are specific to the identification and manipulation of files.

```
interface FileDOb_ID : DirectoryDOb_ID
  {
    attribute string fname;
    attribute long sliceStart;
    attribute long sliceSize;
  };
```

Figure 6.12: *The File Data Object Identifier interface*

The FileDOb_ID provides the data object identifier for files(Figure 6.12). It provides attributes for identifying the directory(fpath in DirectoryDOb_ID) and file name. Optional is attributes for specifying an internal segment of a file to retrieve.

Extensions to the DataObjectManager interface provide manipulation operations to data. FileDObS gives two manager interfaces to access file data. The FileDObManager(Figure 6.13) is based on the Java language's java.io.RandomAccessFile class and allows movement around the file as well as the reading/writing of CORBA data types.

```
interface FileDObManager : DataObjectService::DataObjectManager
  {
    long getFilePointer() raises(IOException);
    void seek(in long offset) raises(IOException);
    long length() raises(IOException);
    void setlength(in long fileLength) raises(IOException);

    octet readByte() raises(EndOfFile,IOException);
    Bytes readBytes(in long NoOfBytes) raises(EndOfFile,IOException);
    char readChar() raises(EndOfFile,IOException);
    string readString() raises(EndOfFile,IOException);
    boolean readBoolean() raises(EndOfFile,IOException);
    short readShort() raises(EndOfFile,IOException);
    unsigned short readUShort() raises(EndOfFile,IOException);
    long readLong() raises(EndOfFile,IOException);
    unsigned long readULong() raises(EndOfFile,IOException);
    float readFloat() raises(EndOfFile,IOException);
    double readDouble() raises(EndOfFile,IOException);

    void writeByte(in octet aByte) raises(IOException);
    void writeBytes(in Bytes someBytes) raises(IOException);
    void writeChar(in char aChar) raises(IOException);
    void writeString(in string aString) raises(IOException);
    void writeBoolean(in boolean aBoolean) raises(IOException);
    void writeShort(in short aShort) raises(IOException);
    void writeUShort(in unsigned short aUShort)raises(IOException);
    void writeLong(in long aLong) raises(IOException);
    void writeULong(in unsigned long aULong) raises(IOException);
    void writeFloat(in float aFloat) raises(IOException);
    void writeDouble(in double aDouble) raises(IOException);
  };
```

Figure 6.13: *The File Data Object Manager interface*

The second manager interface(Figure 6.14) to manipulate files provides a StreamIO interface to files. The StreamIO interface allows CORBA objects to write or read their state to and from a file, therefore providing a simplistic form of persistence.

```
interface StreamIOFileDObManager
      : DataObjectService::DataObjectManager
  {
    CosStream::StreamIO getStreamIOInterface();
  };
```

Figure 6.14: *The StreamIO File Data Object Manager*

The BLOb(Binary Large Object) interface(Figure 6.15) is the component for carrying file data through a stream. The file data is held as a sequence of octets(BLOb_data) and data is put into and taken out of the Streamable using the *add_slice/take_slice* operations.

```
typedef sequence<octet> Bytes;

//BLOb interface for private use of FileDObManager

interface BLOb : CosStream::Streamable,
                  CosLifeCycle::LifeCycleObject
  {
    attribute Bytes BLOb_data;
    attribute long sliceStart;
    attribute long sliceSize;

    void add_slice(in Bytes part_of_file) raises(IOException);
    Bytes take_slice() raises(EndOfFile);
  };
```

Figure 6.15: *The Binary Large Object interface*

6.4.2 Implementation of the File Data Object Service

The implementation of the File Data Object Service(FileDObS) involves a Java client side and C++ server side. Hence the FileDObManager and StreamIOFileDObManager are implemented in Java and the FileDObServer in C++. All other interfaces/components are implemented in both languages.

The Java based components use the JavaIDL[JIDL] ORB from SUN Microsystems and the C++ side uses COOL-ORB[Chorus 97] from Chorus systems. Communication between the ORBs is easily possible due to the fact they both use IIOP as their native communication protocol. IIOP is the standard inter-ORB communication between ORBs on a TCP/IP network, thus inter-operability between the ORBs and the components using them is seamless.

A task necessary to be completed before implementing the Java client FileDObS components is the porting of the client side components of the Stream Tunnel Service(STS) to the Java language. This task was easier to carry out than was first thought, due to the closeness of C++ and Java languages. If a C++ program is fully written in an object-oriented style of only classes and methods, then porting it to the pure object-oriented language of Java is fairly straightforward. C++ classes and methods can be directly translated into Java classes and methods by copy & pasting code from one to other, where sometimes the only conversion needed was the keyword 'public' being added to the method header.

Although porting the components between languages was effortless, a problem was found in the inter-operability between the two ORBs. The intended method of transferring stream data between the ends of stream was to use the ORB octet sequence method. However, when carrying out this transfer, the Java ORB caused an exception to requesting an octet sequence from the C++ server. This is caused by one of the ORBs not fully complying with the IIOP standard for passing sequences of octets, thus either the COOL-ORB is packing the data wrongly or JavaIDL reading the data incorrectly. Suspicion for this fault leans towards the COOL-ORB due to it being a very early ORB product relative to the maturing of CORBA specifications. Except for this problem, no other inter-operability problems were found between the ORBs.

The solution to this problem is to use the alternative stream method of data transfer using sockets. Java provides sockets functionality in a set of standard Java classes within the java.net package. Sending data through a socket connection between the two differing language StreamChannel components worked well and is the data transfer method used for all stream communication between Java and C++ stream components.

Implementation of the File Data Object Service components(Figure 6.16) was accomplished without any major complexities. The FileDObS interfaces were simply implemented by components. These components also carried out their roles that are specified in the Data Object Service, so that the file Data Object Server read/wrote from/to its local file system and produced/consumed Streamable BLOb components for pushing/pulling across a stream. The

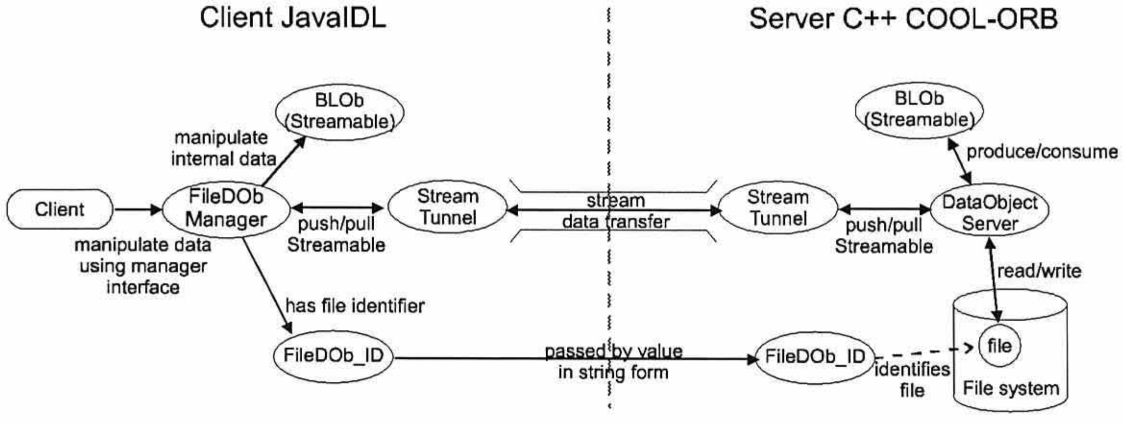


Figure 6.16: *An overview of the File Data Object Service Components*

FileDObManager types handled BLObs on the client side, allowing clients to access the internal data using the manager's interface. FileDOb_IDs are used to identify files for creation, retrieval, storage and deletion. Also, the associate set of components is implemented for the directory data objects that contain a listing of a directory's contents.

6.4.3 CORBAFileView Application

CORBAFileView is an application demonstrating the utilisation of the File Data Object Service. CORBAFileView is a Java application and uses the client Java FileDObS components. The application demonstrates the two fundamental uses of FileDObS, which are the access and manipulation of directories and files. Demonstration of these uses includes the following :-

- creation and deletion of directories
- graphical navigation of directories
- graphical selection of files to retrieve
- viewing of image files
- editing of text files
- storage of text files
- graphical viewing of specific CORBA objects

Figure 6.17 & 6.18 shows example screenshots of CORBAFileView being used to view an image and to edit a text file. The CORBAFileView window is separated into two panes. The left panes allows the navigation of the directory structure and selection of files. While the right pane shows the contents of the file retrieved. It is in the right pane that image files can be viewed and text files can be edited. The menu bar provides options to create/delete directories, create/delete files and exit.

The graphical components that make the application are standard Java classes provided by the Java AWT package[JDK]. For example, there exists a List class for directory contents, a TextArea class for text editing, an Image class for displaying of images. CORBAFileView creates instances of these classes and initialises their state using data retrieved through the DirectoryDObManager/FileDObManager. CORBAFileView reacts to users by responding to events from its graphical components. Responding to events such as double-clicking a

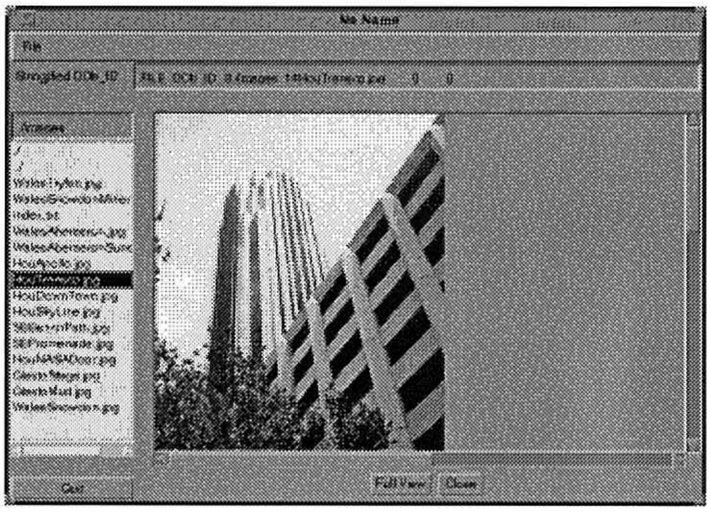
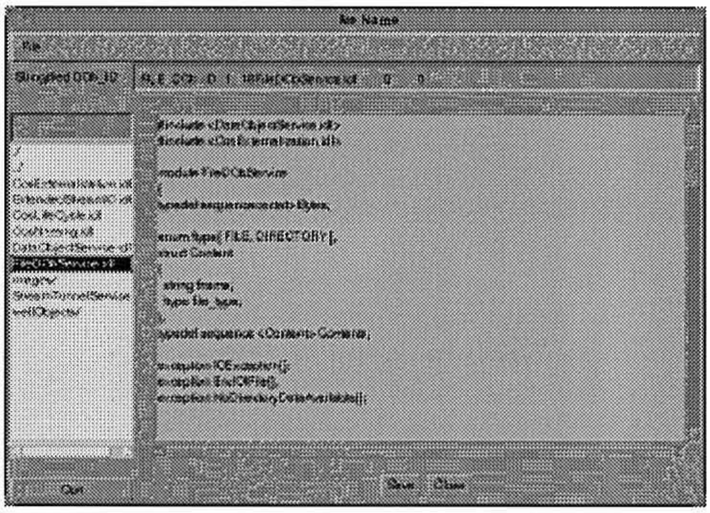


Figure 6.17: *Screenshot of CORBAFileView displaying an image file*



```
#include <DataObjectService.idl>
#include <CosExternalization.idl>

module FileDObService
{
typedef sequence<octet> Bytes;

enum ftype{ FILE, DIRECTORY };
struct Content
{
  string fname;
  ftype file_type;
};
typedef sequence <Content> Contents;

exception IOException{};
exception EndOfFile{};
exception NoDirectoryDataAvailable{};
```

Figure 6.18: *Screenshot of CORBAFileView editing a file*

directory of the list causes CORBAFileView to retrieve and display the contents of that directory, or clicking the text editor save button stores the text file. This demonstrates how graphical interfaces can be layered on top of distributed services to indirectly provide their powerful facilities to users.

CORBAFileView also demonstrates the use of the StreamIO interface provided by StreamIOFileDObManager to externalise/internalise a CORBA object state to and from a file, providing a simplistic form of persistence. The CORBA object's state is graphically displayed(Figure 6.19) allowing the user to change any attributes of the object and to have it stored again. The CORBA objects are loosely based on a few entities taken from the Epicentre data model (the IDL interfaces are shown in appendix D). This small object model represents a common E&P feature that belong to a 'well' entity that has a location and contains multiple

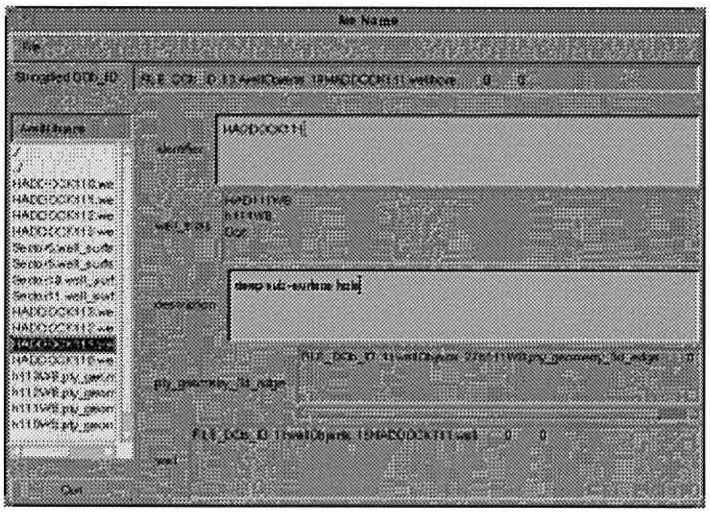


Figure 6.19: *Screenshot of CORBAFileView displaying the persistent state of a wellbore CORBA object*

'wellbore' entities, where a 'wellbore' has attributes, water depth and its three dimensional path below the surface.

The interfaces inherit the Streamable interface so that the implementation must support the *externalize_to_stream* and *internalize_from_stream* operations. These operations use the StreamIO interface to save or initialise the object's state to/from a file. References between the objects are held as a DOb_ID_String attribute type. The internal data of this type identifies the file that contains the referenced object. Double-clicking the graphical representation of an identifier responds in CORBAFileView fetching the object state file and displaying the attributes of the object.

6.4.4 Security and Reliability Improvements to the File Server

CORBAFileView was designed for public demonstration, so that the Java client side could be downloaded or embedded in a web page and used outside the university domain. Thus, the Java client side connects with the C++ COOL-ORB file server located on a university server, and navigates its directories and retrieves its files. Making the FileDObServer available to multiple external clients increases the importance of the security and reliability of the service. The improvements made to the service are discussed next.

Opening a server process up to public access throughout the internet leads to the opportunity for abuse of the open access to gain further access to the private university resources. Hence, the server side of the File Data Object and Stream Tunnel services had to be scrutinised for possible security weak points. One weak point that was found is that the size of parameters being received by the server was not checked. This checking is necessary as a corrupt client could push a large amount of data into the server as a parameter of a request causing the server process to crash. That could open up additional security breaches. This problem was solved by checking the size of all incoming parameters to see if they were too large for what they were representing.

An application level security problem is the fact that the FileDObServer is directly accessing the server's file system, thus external clients have to be prevented from wandering around the file system and stay below a designated directory. To enforce this, all requests for directory paths is taken as an offset to the designated directory and the directory path is checked to make sure the directory paths do not go above the designated directory. Another security precaution is that the FileDObServer is put into read-only mode, so that directories and files cannot be created/deleted and files cannot be edited and stored.

Another problem concerned with opening the FileDObServer to public access is that it executes on a public university server, therefore the FileDObServer must be prevented from overloading the server. A means of achieving this is to limit the number of clients that can simultaneously access the FileDObServer. However, this is not as easy as it first appears, due to the fact that the FileDObServer has no way of identifying clients making requests of it through normal ORB mechanisms. Consequently, it cannot refuse the requests of some clients and carry out the requests of others as it has no way to identify the clients it is serving. The answer to this is to limit the number of streams allowed to be setup between the server and its clients. If a client cannot create a stream, then it cannot request files from the FileDObServer. Although this is not a perfect solution, it will limit the amount of work the FileDObServer performs in moving file data. The FileDObServer implementation was limited to ten stream connections.

However, limiting the number of streams increases the need for a solution to another related problem. This problem involves the possibility of a client not closing a stream before it terminates. This could be due to the client application crashing or a developer forgetting to close the stream. This could result in the number of stream connections running out preventing new streams from being created. The solution is for streams that have not been used for certain period of time to be automatically closed by the server, if the period of time is long enough to indicate that there is a high probability that the client has terminated without closing its stream. This is a form of garbage collection, where stream connections are regularly examined to see if they have been used within the time limit, and if they have not been used then they are closed. Garbage collection takes place at regular intervals, hence the implementation of it is on a independent thread of execution that periodically wakes up, checks when the streams were last used, closes any unused streams and sleeps again. As a consequence of this garbage collection, the StreamTunnel has to create the garbage collector thread and the StreamTunnel component has to be locked when the garbage collector is awake and running.

6.5 Similarities and Differences between the Data Object Service and the Persistent Object Service

The Data Object Service(DObS) has many similarities to the failed Persistent Object Service(POS)(see section 3.3.1)[Ball 98], but also has many differences that are the result of what has been learnt from the design failures of POS.

The Data Object Service has a similar purpose to the Persistent Object Service in that they provide access to persistent data resident in heterogeneous datastores within the CORBA environment. Both services specify interfaces to identify, create/delete, retrieve/store and transfer data. There is a great deal of commonality between the components of the services, and an interface in one service has an equivalent interface in the other carrying out an equivalent role. The equivalent interfaces/components and their roles are shown in the following table.

Data Object Service Interfaces	**Persistent Object Service Interfaces**	**Role**
Data Object Identifier (DOb_ID)	Persistent Identifier (PID)	Identify data that is to be managed or manipulated.
DataObjectManager	Persistent Object Manager(POM)/Persistent Object(PO)	Retrieve/store identified data and to manage and manipulate it.
DataObjectServer	Persistent Data	Retrieve/store data from its

	Store(PDS)	support datastore and transfer it the client components using a data transfer mechanism.

Both services also use the concept of extending their interfaces to provide the functionality to support different types of data. For example, DOb_ID and PID are extended to identify different types of data, while the DataObjectManager and PO are extended to manipulate different types of data.

There is a conceptual difference between the client components of the services i.e. DataObjectManager and the POM/PO. This difference is due to POS's principal objective, which is to provide persistence to CORBA objects. Thus, POS provides a mechanism to retrieve and store the persistent state of a PO object. Hence, there is a one-to-one relationship between a PO object and a unit of data. DObS works on a more flexible way of a manager that contains all the functionality to retrieve/store and manipulate data. The benefit of a manager manipulation style is that a single manager instance can handle many items of data. Also, it separates user CORBA objects from the functionality required to interact with datastores and to transfer data, thus making them simpler. The POS manager - the POM does not provide any of this functionality, but is only used to route requests to a target datastore.

The most significant distinction between the two services is that DObS specifies a single data transfer mechanism(i.e. STS) that is highly integrated into the service. POS, on the other hand, suggests a number of possible data transfer protocols. Conformance to these protocols is left to the internal behaviour of POS components. This causes the data transfer protocols to be weakly integrated into POS. Weak integration like this results in non-conformity between implementations of POS.

Another benefit of DObS integration with STS is that it is more efficient than any of the POS data transfer protocols. The POS protocols of Direct Access and Dynamic Data Object are intrinsically inefficient due to the way they transfer data. This method entails a remote request to retrieve every attribute of an object. There is a large performance penalty inherent in remote requests, and consequently the POS protocols will become unusable for even small amounts of data. STS allows all data to written into a single stream within local memory, and the stream's data is then transported in bulk to the destination. The use of streams in POS has been suggested by one of the architects of POS in [Sessions 96], but no formal integration of a stream mechanism into POS was suggested.

A problem that the use of STS also avoids is the deadlocking problem of POS, whereby a deadlock situation occurs due to an object requesting an operation of another object which is blocked or does not run on an object adaptor. STS avoids this by driving all data transfers on the client side of the stream, as all communication is in the direction of client side to server side only.

The Data Object Service(DObS) is also efficient with its use of identification of data. The Data Object Identifier(DOb_ID) is passed by value in a string form, so that the DOb_ID object can be reconstructed by the server. POS passes the equivalent component, the PID by reference, so that remote requests are required to query the PID information, hence performance is degraded. Passing the PID by reference is also a major design fault of POS. In CORBA, if an object is to service requests as would be the case to query the PID information, then that object must run on an object adaptor. If the PID object is resident within the process space of a client application, then the client application must run an object adaptor. This is impracticable as applications generally have synchronous execution and are singularly threaded.

The Data Object Service and the Stream Tunnel Service provide more efficient and unambiguous service than the Persistent Object Service. DObS has been proven to work within the implementation of the FileDObS, while POS has no satisfactory implementations due its inherent design flaws and ambiguities.

6.6 Lessons Learnt from Implementing CORBA Services and Applications

This section discusses the lessons that have been learnt from carrying out the implementation of these CORBA services and applications. It puts forward advice that can be applied to general application development and that any developer starting developing CORBA applications should be concerned with.

A critical lesson learnt is to very carefully choose the system software which developed applications will depend upon. System software is the software which an application becomes dependent on for some facility provided by the software. Examples of system software and the facilities they provide are data storage in databases, distributed communication by ORBs and graphical interfaces by graphical component libraries. The factors that should be weighed up in deciding on a piece of system software is the need for portability against the need for specialised features of the system software.

Portability of application code over differing variants of the system software is an important consideration. Applications can become locked to a specific features of a specific vendor's system software. To port such applications to other vendor's system software may require much effort, sometimes even impossible.

Portability problems became apparent in this project with the need to port code to ORBs other than COOL-ORB. COOL-ORB became undesirable due to problems such as no support for multi-threading, IIOP interoperability problems and does not compliance with later CORBA specifications concerning ORB, BOA and CORBA object initialisation. Also, COOL-ORB was withdrawn by Chorus Systems so that development could not take place on other platforms due to COOL-ORB not being available for them. If COOL-ORB had complied with the CORBA standards and the developed application code was written to these standards, then porting the code to other ORBs would be relatively effortless. Thus, the lesson learnt was to choose system software that complies with standards set for that area.

An opposing force to choosing system software based on portability is the need to choose system software that is feature-rich. Features provide additional functionality, enhancing the quality of applications and possibly improving their performance. However, software providing such features are generally proprietary to the specific vendor's software and are not part of a standard. For example, an ORB which is feature-rich is Iona's Orbix. Orbix provides many useful and powerful features such as callbacks, filters and smart proxies[Baker 97] to name a few. But none of these features are standard, therefore cannot be used in conjunction with other vendors' ORBs. Using features such as these will lock applications and whole systems to a specific vendor's ORB.

The need for the additional feature of multi-threading became a problem in implementing server components of the services. As COOL-ORB did not support multi-threading, low-level operating system mechanisms (e.g. threads and semaphores) had to be employed to serve the same purpose, thus dramatically increasing the complexity of applications. This was another lesson that was learnt, which is to try to only use facilities that are inside the paradigm that the majority of an application uses. The threads and semaphores used are not based in object-orientation, thus much complexity was introduced to make use of them from the object-oriented application.

Feature-rich ORBs usually provide thread class libraries that encapsulate the use of native platform threads in an object-oriented manner. Therefore, using mechanisms that provide high-level control of threads greatly simplifies developing multi-threaded applications. However, there is no set standards for thread control, hence each thread class library will be proprietary to the specific vendor's ORB and becomes another possible source of vendor lock.

Choosing system software such as an ORB is a critical decision that must be taken before any development work is carried out. To make the best decision, the need for portability has to be foreseen as well as what features will be required by applications. These two opposing factors have weighed against each other and the result will influence the system software employed.

One of the great strengths of CORBA is its ability to provide interoperability that is of transparent of location, platform and ORB type. Therefore, to a client of a CORBA object, these transparency factors are of no importance. For example, the CORBA object could be resident on a mainframe server somewhere on the internet or in the same process as the client, it has no bearing on the way the client inter-operates with the CORBA object. The benefit of this transparency is that it aids the scalability of applications, whereby objects can be redistributed across different machines to increase performance with no alteration needed to actual application code. Scalability is easy due to the CORBA transparency nature enabling low coupling between objects i.e. there is a low dependency between components. The only dependency between inter-operating components is their IDL interfaces, which is a well defined dependency. Thus, to make applications as scalable and flexible as possible, every component should inter-operate through an IDL interface, avoiding language specific mechanisms such as function calls, method invocations etc.

An area where it is not possible to avoid language specific mechanisms is in the creation of CORBA objects. Creating a CORBA object involves using a language specific instance creation construct such as C++'s 'new', causing a high dependency between the creating component and the created component. This high dependency results in a very rigid association between the components, preventing components either from being scaled and making it very difficult for either component to be altered and extended without effecting the other.

The OMG solution to this object creation problem is in the LifeCycle service. The LifeCycle service states a technique in which objects are created by Factory objects. A Factory object encapsulates all the complexity dealing with creating an object including creating an instance of it, registering and initialising it. This technique disconnects applications from object creation, lowering the dependency between the application and the creation of the object. Use of Object factories is a valuable technique to try and solve this problem, but the LifeCycle service does not go far enough. The LifeCycle model of creating CORBA objects is not a completely satisfactory solution due to complexities such as how Factory objects are created and how they are found.

The LifeCycle service caters for the finding of Factory objects with the FactoryFinder interface. The FactoryFinder object returns a list of Factory objects to a client that are associated with some key information. The problem with the LifeCycle service's FactoryFinder is that there is no standard way to register Factory objects with the FactoryFinder. To hard-code registered Factory objects in a FactoryFinder is not a very flexible solution, therefore a source of non-portability. A better solution is to dynamically register Factory objects with a FactoryFinder. This can be achieved by the FactoryFinder being extended to offer a registration interface. This allows a Factory object to dynamically register itself upon creation, consequently letting clients find it through the *find_factories* operation. Again, this is an example of lowering the dependency between components so that all inter-operability is through IDL interfaces with no reliance on language-specific mechanisms.

The second complexity that the LifeCycle service does not provide a solution to is the creation of actual Factory objects. In fact, there is no avoiding language-specific creation mechanisms to do this, as somewhere in an application the Factory objects have to be created. However, the effect on scalability can be minimised by keeping all Factory object creations in a single location in a program and not spreading it throughout different sections of a program. Therefore, if the program did need to be altered for scalability purposes or for some other purpose, then the effect of altering the Factory object creation section would be minimal to the rest of the program. The ideal situation is to have all Factory objects created, initialised and registered with a FactoryFinder, before an application starts to perform its task. This way all Factory objects are ready to be found and used by the rest of an application.

These suggestions concerning creating CORBA objects and their Factories came from experience in needing to create the objects making up the services described in this chapter. Hard-coding the creation of objects proved to be cumbersome and complex when it came to amending or extending the code, therefore the LifeCycle Factory model was employed to create objects. The FactoryFinder was also hard-coded, proving problematic when altering code and adding new Factory objects.

In general, a common object creation policy should be employed across an application that decreases the dependency between parts of the program, so that ease in changing code, extending code and scalability is maintained.

6.7 Summary

The Data Object Service builds upon the Stream Tunnel Service providing interfaces to manage data objects, which includes identifying, creating/deleting and retrieval/storing of data objects. The combination of these services provides a framework of interfaces to access and manipulate simplistically structured persistent data.

The Data Object Service provides create, retrieve, store and delete operations on identified data objects. Data objects have to be transported between the datastore process that is storing them to the client application for manipulation. This data transportation is provided by the Stream Tunnel Service. DObS strongly integrates STS into the DObS interfaces. DObS has many similarities to that of the failed OMG Persistent Object Service(POS), but DObS has been designed to avoid many of the design failures of POS, particularly the specification of a single data transport mechanism i.e. STS. DObS is only an abstract service, hence it must be extended to provide identification and manipulation operations for specific types of data objects and datastore types. An example of such an extension of DObS is the File Data Object Service which allows the management and manipulation of files and directories. CORBAFileView is a GUI application that demonstrates the use the File Data Object Service by allowing the navigation of directories and the selection of files to retrieve. CORBAFileView can be used to view image files, edit text files and also demonstrates the use of the File Data Object Service to provide persistence to CORBA objects.

Many lessons have been learnt from implementing these CORBA applications and services. One of the major lessons is to very carefully choose system software e.g. an ORB, so that portability of application code that is dependent on the software is maintained. Opposing the portability issue is the need for specialised features of the system software that enhances the quality and performance of the applications, but are proprietary to a particular product. These issues should be carefully weighed up upon selection of system software by predicting the need for portability and the judging what proprietary features that will be needed.

Another valuable lesson was to make all components comprising an applications to interoperate through IDL interfaces, therefore minimising inter-dependency and reducing dependency on language specific inter-operation mechanisms. Reducing dependencies between components increases the flexibility and scalability of applications. An area where language specific mechanisms cannot be avoided is in the creation of CORBA objects. However, this language dependency can be reduced by using the LifeCycle model of Factory objects and dynamically registering Factory objects with FactoryFinder objects.

The Data Object Service and the Stream Tunnel Service is fine for accessing and manipulating data objects of simplistic structured data such as files. But for more sophisticated structured data like stores of complex structured objects having reference and inheritance relationships, then more advanced functionality is needed than DObS provides. This functionality will be provided by the Persistent Data Access Service(PDAS). However, PDAS has a need for the transport of data between the client and server sessions of the service, which the union of STS and DObS(and a specialisation service of it) can easily fulfil.

Chapter 7

The Persistent Data Access Service

7.1 Introduction

This chapter presents an overview of the Persistent Data Access Service(PDAS). This service is the decisive solution to the set requirements to provide a framework of interfaces to allow access and manipulation of persistent data. The persistent data is resident in heterogeneous datastores which are accessible in the distributed environment, and datastore types will have proprietary data definition models. The purpose of this service rather than a direct method of data access with proprietary datastore interfaces is to insulate applications from their proprietary nature, therefore lowering the dependency of application code on the data access interface to the datastore.

The Persistent Data Access Service brings together the high level design features discussed in chapter 4, as well as the services previously designed and implemented i.e. the Stream Tunnel Service and the Data Object Service. These services are extended for managing and transporting the complex structured entity data that PDAS is manipulating.

The most significant high level design feature was the provision of a standard data definition model. This standard data definition model is realised in the form of an object model called the

Meta-Schema model. All data manipulated by PDAS conform to this model, hence all data models defined using proprietary data definition models must be mapped to the Meta-Schema model providing a common view of the structure of data(Figure 7.1).

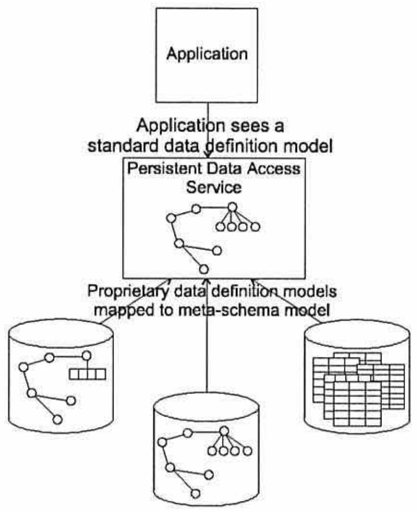

Application
Application sees a standard data definition model
Persistent Data Access Service
Proprietary data definition models mapped to meta-schema model

Figure 7.1: *The Meta-Schema model provides a common view to the proprietary data models*

The meta-information provided by the Meta-Schema model is crucial to application components to enable them to dynamic handle entities. For components to retrieve this meta-information, the meta-schema model has a direct IDL translation(Appendix E), hence CORBA objects are instantiated containing the meta-information.

Another high level feature is the use of a session component to provide the following:- access to the persistent data, representation of a connection to a datastore and integration with the transaction and concurrency services. The location of the session determines whether the data is manipulated locally in the client process or remotely in the server process.

For the data to be locally manipulated, the data has to be transported to the client process from the datastore server process i.e. the server session. Generally, the data transport mechanisms that datastores like relational DBMS's or object-oriented databases use to move data across the distributed environment are proprietary to the specific datastore product and closed to external developers. PDAS's data transport mechanism is an open mechanism based on the Data Object Service and the Stream Tunnel Service. The alliance of these two services acts as an object bus transporting entity data between the distributed client and server PDAS sessions(Figure 7.2).

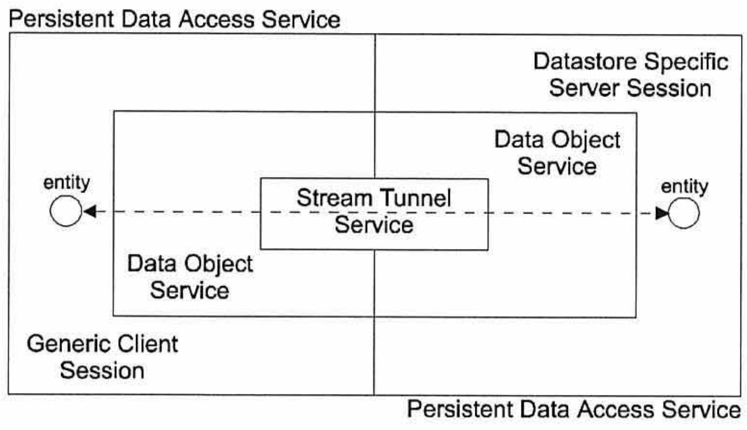

Persistent Data Access Service
Datastore Specific Server Session
Data Object Service
entity
Stream Tunnel Service
entity
Data Object Service
Generic Client Session
Persistent Data Access Service

Figure 7.2: *The Stream Tunnel Service and Data Object Service provide an object bus transporting entities between client and server sessions*

The objective of the service is to free client applications from their dependency on the proprietary datastore containing its data. Hence, to enforce this objective the client side to the Persistent Data Object Service is purely generic, in the sense that it can handle any data from any data model(conforming to the Meta-Schema model) held in any datastore. To provide this generic characteristic, all inter-operability between the client and server sides of the service has to be standardised, including accessing, managing and transporting the data. This standard data access and management facility is provided by an extension of the Data Object Service that is geared towards managing the structured entity data.

The standard transportation mechanism of structured entity data is provided by the Stream Tunnel Service(STS). However, STS and more specifically the Externalization service, only provides for low-level literal types to be read/written from/to a stream. PDAS requires higher level structured entity types to be read/written from/to the stream to maintain the structure of entities in their serialised form. Thus, a protocol that formalises rules about how entities and their attributes are held in the stream is specified. This is a powerful concept as it allows the transport of complex structured entities in a distributed carrier mechanism i.e. the stream, providing a mechanism for the transport and exchange of highly structured data(Figure 7.3).

The design factors that also make the transport of complex entity data possible is the use of object identifiers(oids) and the availability of meta-information on the data. Object identifiers provide an independent language/platform reference representation, so that internal datastore references can be represented in the external domain of the PDAS sessions, hence maintaining

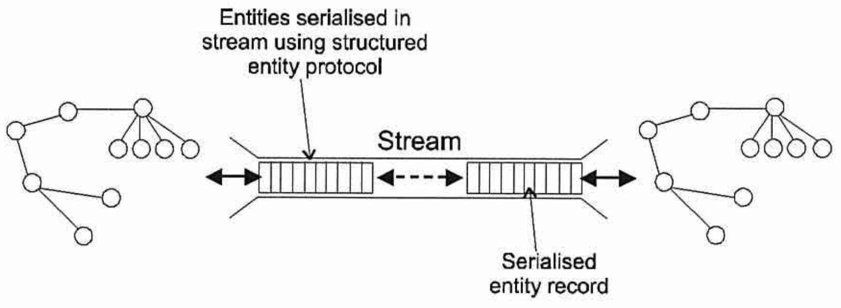

Entities serialised in stream using structured entity protocol
Stream
Serialised entity record

Figure 7.3: *Highly structured entity data can be transported using the stream given a protocol for specifying how entities are held in a stream*

entity relationships. Provision of meta-information from the Meta-Schema model is a crucial utility to the service. Using the Meta-Schema model, the structure of entities and their attribute types can be discovered, enabling the dynamic handling of entities. Dynamic handling of entities is important in a number of roles that PDAS has to carry out, including dynamic mapping of data in their proprietary datastore form to a standardised form, (de)serialising the data to/from the stream and enabling the dynamic manipulation of entity attributes.

This chapter presents how these concepts and design features are integrated into the Persistent Data Access Service(PDAS). Firstly, an overview of the IDL interfaces and the responsibilities of the components implementing these interfaces is presented. Next is a short scenario demonstrating how the PDAS components work together to retrieve entities of small data model.

7.2 Modules and Interfaces of the Persistent Data Access Service

The Persistent Data Access Service is separated into three IDL modules, where each module contains a set of interfaces and performs a specific role in the service. The modules are shown in figure 7.4 and have the following roles and responsibilities :-

• The Persistent Data Access Service(PDAS)	Interfaces of this module are the primary interfaces that client applications of the service interact with. The interfaces include the creation of sessions, actual sessions and an interface to manipulate the attributes of entities.
• The Entity Data Object Service	These interfaces extend the Data Object Service and the Stream Tunnel Service to provide a mechanism for the access and

transport of complex structured entities.

- The Persistent Data Access Service Server (PDASServer) — This is the server side to the service that includes the actual ServerSession and a way of creating a ServerSession. The most significant role the ServerSession performs is to access the supported datastore and map accessed data to and from the standardised format.

Also, implementation of these interface will require access to meta-information from the Meta-Schema model. The Persistent Data Access Services modules and interfaces are discussed in further detail in the following sections. The modules and interfaces of the service are shown in full in appendix E.

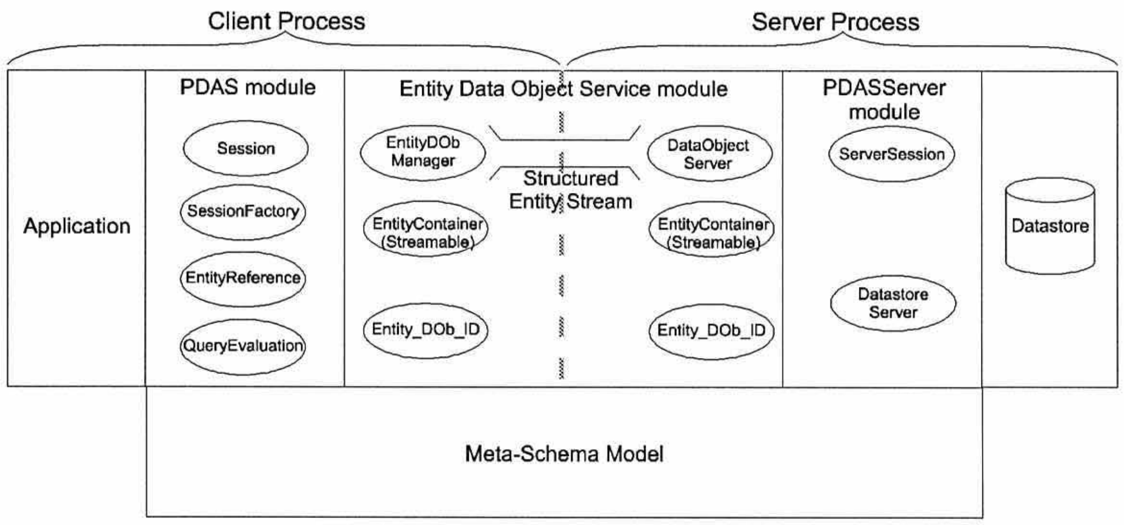


Figure 7.4: *Modules of the Persistent Data Access Service*

7.2.1 The Persistent Data Access Module

The interfaces of the Persistent Data Access Service(PDAS) module(appendix F) are the application's only contact with the service. All other interfaces are concerned with the internal workings of the service and have no relevance to the application. The PDAS module interfaces allow the application to create a session, gain references to entity instances and execute queries to gain initial entities.

```
interface Session      : CosConcurrencyControl::LockSet
{
   EntityReference create(in string entity_type_name)
           raises(NotFound,CreateFailure);
   EntityReference copy(in EntityReference source_entity)
                       raises(CreateFailure,ExactCopiesNotSupportedByModel);
   void delete(in EntityReference entity) raises(DeleteFailure);

   void save_all_updates( ) raises(StoreFailure);

   MSModel::msSchema get_schema_model( );
   RetrievalModel::RetrievalMaps get_retrieval_maps( );

   CosQuery::QueryEvaluator get_query_evaluator( );

   Object bind_object(in EntityReference er,
           in CosLifeCycle::FactoryFinder finder)
                       raises(CosLifeCycle::NoFactory);

   EntityReference find_entity_by_oid(in long oid) raises(NotFound);

   void close_session( ) raises(TransactionInProgress);
};

interface NonTransactionalSession : Session
{
};

interface TransactionalSession : Session,
                                 CosTransactions::TransactionalObject
{
};
```

Figure 7.5: *The Session interface and its non-transactional and transactional extensions*

The Session interface(Figure 7.5) is the application's principal interface to the service's functionality. The Session with its transactional extension also supplies the client side integration with the transaction and concurrency services. Operations of the Session provides the following :-

- Basic create, copy and delete of entities.
- Access to the Meta-Schema model describing the entities being manipulated.
- Access to the Retrieval Map model describing sub-groups of entities for simultaneous retrieval and storage.
- Save updates to entities.
- Access to the query service.
- Binding of a static CORBA objects to entity objects.
- Gaining entities using session dependent oid values.
- Close a session.

The *create* operation of the SessionFactory(Figure 7.6) interface is used to create a Session as well as setting up and initialising a ServerSession for the Session to connect to. The attributes of the *create* operation indicate a datastore server, the name of a datastore the server

```
interface SessionFactory
{
   Session create(in PDASServer::DatastoreServer datastore_server,
                     in string datastore_name,
                     in boolean transactional_session,
                     in CosConcurrencyControl::lock_mode initial_lock_mode,
                     in CosPropertyService::Properties initialisation_attributes)
                     raises(CreateFailure);
};
```

Figure 7.6: *The SessionFactory interface, used to create Session objects*

```
interface EntityReference
{
   boolean is_same(EntityReference other_entity);
   boolean is_kind_of(string entity_type_name);

   MSModel::msEntity get_entity_type();
   MSModel::msType get_attribute_type(in string attr_name)
          raises(InvalidAttribute);

   long get_entity_session_oid( );

   void save_updates( ) raises(StoreFailure);

   void release( );

   // Operations to set the values of entity attributes
   void set_String(in string attr_name, in string s)
          raises(InvalidAttribute,IllegalCast);
   void set_Double(in string attr_name, in double d)
          raises(InvalidAttribute,IllegalCast);
   :                                                    :
   : Further set meta-schema attribute type operations  :
   :                                                    :
   void set_Aggregate(in string attr_name, in CosCollection::Collection c)
        raises(InvalidAttribute,IllegalCast);
   void set_EntityReference(in string attr_name, in EntityReference er)
          raises(InvalidAttribute,IllegalCast,InvalidRefenceAssignment);

   // Operations to get the values of entity attributes
   string get_String(in string attr_name)
          raises(InvalidAttribute,IllegalCast);
   double get_Double(in string attr_name)
          raises(InvalidAttribute,IllegalCast);
   :                                                    :
   : Further get meta-schema attribute type operations  :
   :                                                    :
   EntityReference get_Entity(in stringattr_name)
          raises(InvalidAttribute,IllegalCast,NullReference);
   EntityReference get_Entity_using_map(in string attr_name, RetrivalMap map_name)
          raises(InvalidAttribute,IllegalCast,NullReference);
};
```

Figure 7.7: *The EntityReference interface allows the manipulation of an entities' attributes*

has access to, whether the session is within a transaction, initial lock type and any other necessary initialisation attributes e.g. user name and password.

The Session interface provides the service's management operations. Actual entity instance manipulation is carried out using the EntityReference interface(Figure 7.7).

The EntityReference interface allows manipulation of entity attributes by providing get and set operations for all the entity attribute types. Each get/set operation has to specify the attribute name of the target attribute since the entity is being dynamically manipulated, and has no static interface. The EntityReference interface provides other operations concerning gaining meta-information on the type of entity the EntityReference is representing. The interface also provides management operations such as getting its oid, saving updates and releasing the EntityReference object when it is no longer needed.

An application must firstly obtain an EntityReference object to an entity instance to be able to manipulate it. There are two general ways that an application can obtain an EntityReference. The first is via the query service, the execution of a query returns a collection of EntityReference's representing the results of the query. The second means is by navigating relationships between entities by calling *get_Entity* on a entity reference attribute.

7.2.2 The PDASServer module

The PDASServer module contains server side interfaces to the service. The DatastoreServer provides an operation to create a ServerSession. In general, newly created ServerSessions will run on separate threads of execution, thus many ServerSessions will be concurrently accessing the datastore(Figure 7.8).

The ServerSession and its (non-)transactional interface extensions(Figure 7.9) provide the server-end to a session. The operations provided by the ServerSession are only accessed by the client Session. It is the responsibility of the client Session to get the ServerSession's DataObjectServer and use it to retrieve entities by setting up a stream with the server.

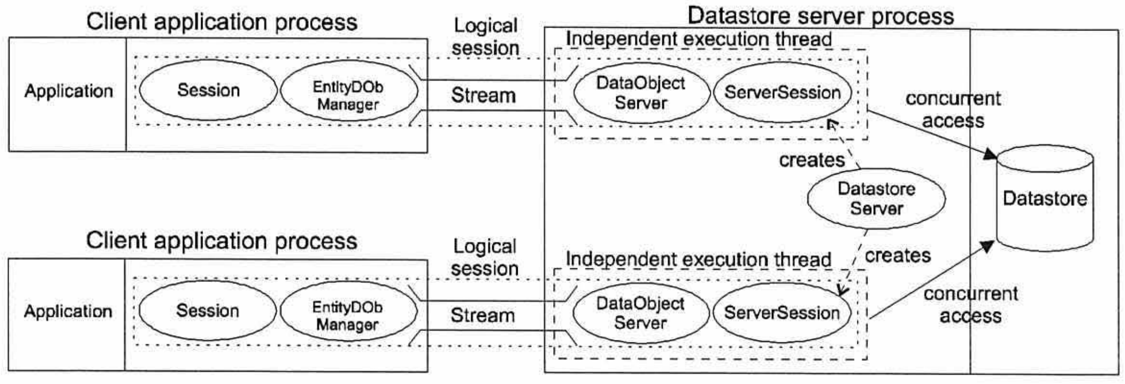


Figure 7.8: *Multiple ServerSessions can be created within the datastore server process that concurrently access the datastore*

```
interface ServerSession : CosConcurrencyControl::LockSet
{
   DataObjectService::DataObjectServer get_data_object_server( );

   MSModel::msSchema get_meta_schema_model( );
   MSR::MetaSchemaRespository get_meta_schema_repository( );

   RetrievalModel::RetrievalMaps get_retrieval_maps( );

   CosQuery::QueryEvaluator get_server_query_evaluator( );

   void close_session( ) raises(PDAS::TransactionInProgress);
};

interface NonTransactionalServerSession : ServerSession
{
};

interface TransactionalServerSession : ServerSession,
                                           CosTransactions::Resource,
                                           CosConcurrency::LockCoordinator
{
};
```

Figure 7.9: *The ServerSession interface and its non-transactional and transactional extensions*

7.2.3 The Entity Data Object Service

The Entity Data Object Service(EntityDObS) is an extension of the Data Object Service to provide access, management and transport of complex entity data. This service is essential to the ability to cache entities in the local process of a client application. Thus, EntityDObS acts as a service for transporting entities between a PDAS client and server sessions. The EntityDObS interfaces are lengthy and complex, so only an overview is given here (the complete IDL interfaces can be found in appendix F).

The type of data objects managed by EntityDObS are entity instances, where an entity instance is made up a collection of attribute values with a system unique identifier(oid). As with all

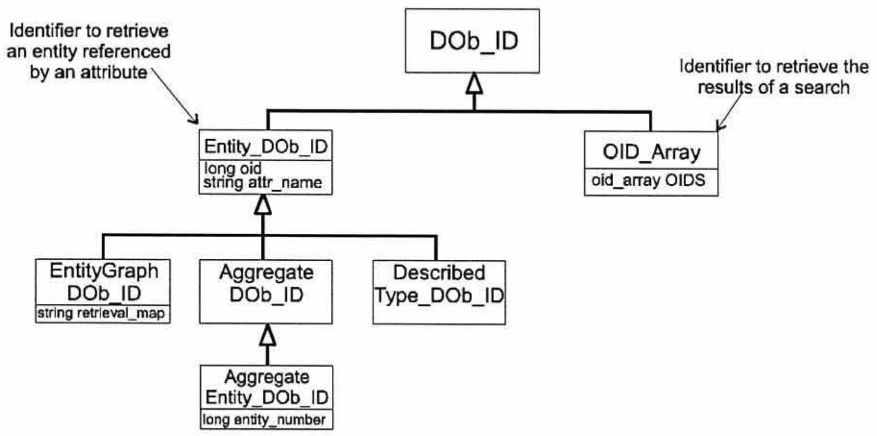


Figure 7.10: *The hierarchy of Data Object identifiers used to identify entities, graphs of entities, aggregate and describe type attributes*

extensions of DObS, the DOb_ID has to be extended to identify the data being accessed. EntityDObS provides the DOb_ID interface hierarchy shown in figure 7.10 for this use. The identifier interface hierarchy allows the retrieval of entities through the results of a search(OID_Array) or through navigation from an entity reference attribute(Entity_DOb_ID). Specialisations of Entity_DOb_ID permit the retrieval of graph of entities and the explicit retrieval of aggregates and described types(i.e. binary data).

Data transported through a stream must by be contained within a Streamable object, where the object knows how to read and write its data from/to the stream. The EntityContainer interface extends the Streamable interface by adding operations to add and take out entities to/from the container. Thus, the EntityContainer can simultaneously contain multiple entities and transfer them through the stream in a single transfer.

The structure of entities has to preserved in their serialised form, so that attribute values written to a stream are correctly read out. Therefore, a protocol has to be specified that defines how entities are held in a stream. The Structured Entity Stream Format(Appendix F) defines this protocol. It specifies how entities and their attribute values are written to a stream including tag values that retain the structure of entities. Figure 7.11 shows an example of the contents of a stream that has had a entity instance serialised to it using the structured entity format. The stream contains all entity identification information and its attribute values. The structure of this information is maintained using tag values that identify what the following value represents.

The Structured Entity Stream Format is another protocol layer on top of the Externalization service's Standard Stream Data Format(SSDF). SSDF specifies the low-level format for

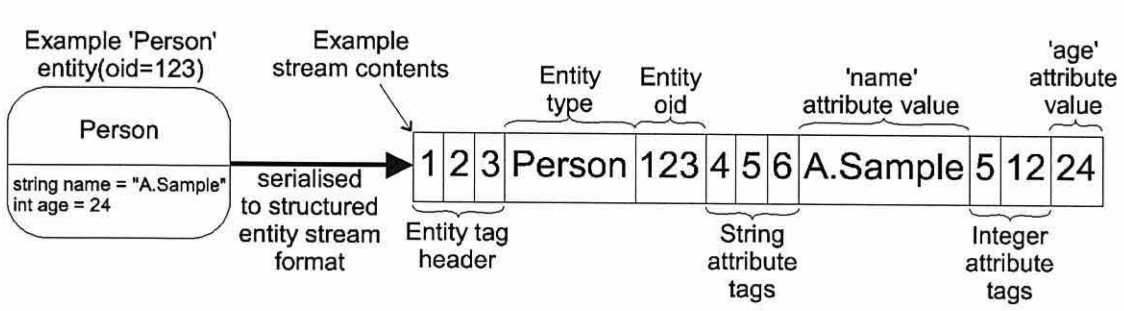


Figure 7.11: *An example of the contents of stream containing a 'Person' entity using the Structured Entity Stream Format*

CORBA data types in the stream. The Structured Entity Stream Format adds a protocol to SSDF that allows the storage of higher level entities. The StructuredEntityStreamIO interface encapsulates the entity format by allowing entities and their attributes to read/written from/to a stream. This achieved by wrapping the Externalization service's StreamIO interface(Figure 7.12).

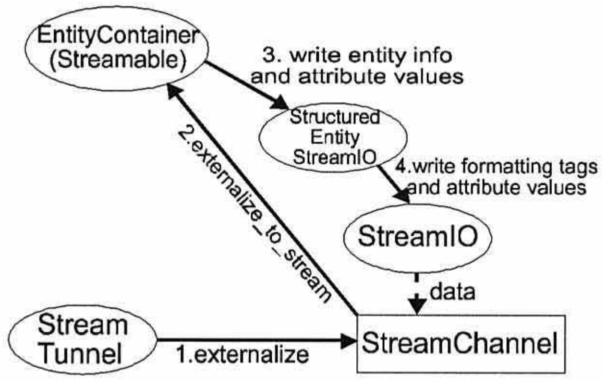

EntityContainer (Streamable)
3. write entity info and attribute values
Structured Entity StreamIO
4.write formatting tags and attribute values
StreamIO
2.externalize_to_stream
data
Stream Tunnel
1.externalize
StreamChannel

Figure 7.12: *The StructuredEntityStreamIO interface wraps the Externalization service's StreamIO interface to allow entities to be read/written from/to a stream using the Structured Entity Stream Format*

The Entity Data Object Service(EntityDObS) extends the Data Object Service, hence the Entity_DOb_ID interface extends the data object identifier. The EntityContainer extends the Streamable interface to hold entities for transport through a stream. The final Data Object Service interface extension is the EntityDObManager interface extending the DataObjectManager. The EntityDObManager encapsulates the transfer and retrieval of entities mechanism allowing the retrieval and storage of entities.

The service also allows for the explicit retrieval of aggregate and described type attributes in separation from their containing entities. These attribute types can be explicitly retrieved, due to the fact that they can be very large. Therefore to retrieve them every time their containing entity is retrieved would be wasteful as the aggregate/described type attribute might not be accessed.

7.2.4 Use of the Meta-Schema Model

The Meta-Schema model(see section 4.4.2) is used to describe the structure of entities including their entity hierarchy and their non-referential and referential attributes. This is achieved by creating instances of the Meta-Schema model that represents the definition of an` entity as described in its native data model. It might not be possible to directly represent the native data model using the Meta-Schema model, therefore some degree of mapping might be necessary to mould the native data model to fit the Meta-Schema model. For example, a data type such as a SQL timestamp or an Epicentre spatial type that are native to a data model would have to be mapped to an entity.

The description of entities provided by the Meta-Schema model enables system components to dynamically handle entities that the component has not been pre-configured to handle. For example, given a segment of memory containing an entity, a component will know the structure of the entity, and can interpret the contents of the memory segment and access the entity attributes. Components that can dynamically handle structured data are superior to static pre-configured components as they are generic i.e. used for any data model and are easily scaleable and flexible to system change.

The information contained in instances of the Meta-Schema model that describe a data model has to be somehow be presented to system components. This is accomplished by representing the Meta-Schema objects as IDL interfaces(Appendix E). CORBA objects implementing these interfaces are created to represent data model meta-information. System components can then query the interfaces of these objects to retrieve the meta-information.

Access to meta-information is a common feature of many persistent data storage systems such as meta-tables in relational databases or a compiled EXPRESS model of data contained in a STEP file. The meta-information is used to present a description of the stored data to applications and to serve as an internal system facility allowing the dynamic processing of described data. There are various components internal to an implementation of the Persistent Data Access Service that will require access to the meta-information. These components shown in figure 7.13 and their need for meta-information is described in the following:-

• Application	The client application might require meta-information of entities to dynamically manipulate data.
• Client PDAS components	The client PDAS components requires meta-information to allow the application to manipulate the attributes of cached entities using EntityReference interface. It is also used to maintain the integrity of the entities, so that the application is prevented from illegal type casting when setting entity attributes.
• EntityContainer	The EntityContainer is the Streamable component, hence reads and writes entities from/to the stream. It is the task of the EntityContainer to break an entity down into its constituent attributes. The names and types of the entities' constituent attributes are found in the meta-information.
• DataObjectServer	The DataObjectServer is the principal component that carries out the retrieval and storage of data. Thus, it has to carry out mapping between the proprietary stored form of the data and a standard entity form that can be added to an EntityContainer. The mapping can be automated by examining the meta-information and finding out the type of each attribute, then perform an automatic conversion between the proprietary types and standard types of the attribute.

The provision of meta-information is crucial to the internal workings of the service, hence there will be frequent accesses to the Meta-Schema Facility to gain meta-information. If the Meta-Schema Facility is a separate system-wide service to the distributed environment then remote requests are required to access the facility. Hence, performance becomes a concern due the low speed of remote requests and the frequency that they will be used by the service. In a perfect distributed environment, local intra-process requests would take the same time as remote inter-process requests. But this is not the situation, so the solution is to move frequently accessed data to the accessing process. Thus, components accessing the Meta-Schema Facility should have a local copy of the data model descriptions stored in the facility.

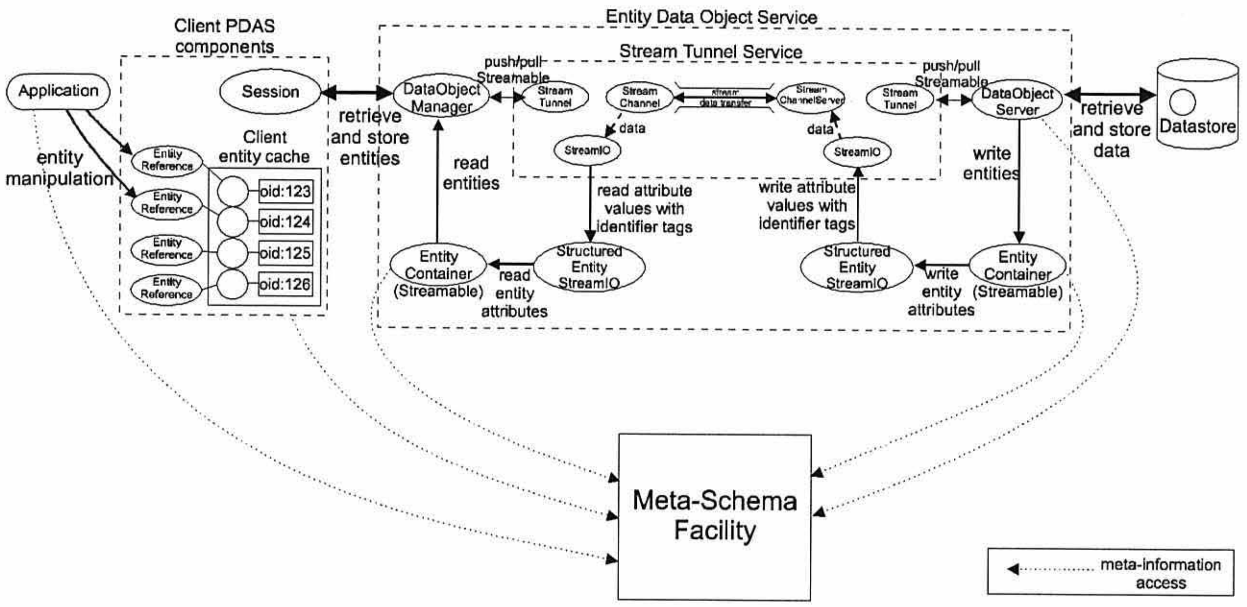


Figure 7.13: *Components of the Persistent Data Object Service that access the Meta-Schema Facility to gain meta-information on entities that are being handled*

Local copies of the data model descriptions can be obtained in two ways: the first method is having a static compiled copy linked with the components at compile time, the second method is to dynamically obtain a copy at run-time. Statically compiling the meta-information into components leads to many difficulties, such as need for recompilation if data models are changed or extended and is also a point of dependency for a generic client side to the service. Therefore, to retrieve a copy of a data model at run-time would the most flexible and generic solution.

Data models are described using CORBA objects that have IDL interfaces representing the object types in the Meta-Schema model. Therefore, to provide local copies of the meta-information, the group of CORBA objects making up a data model has to be copied by value to

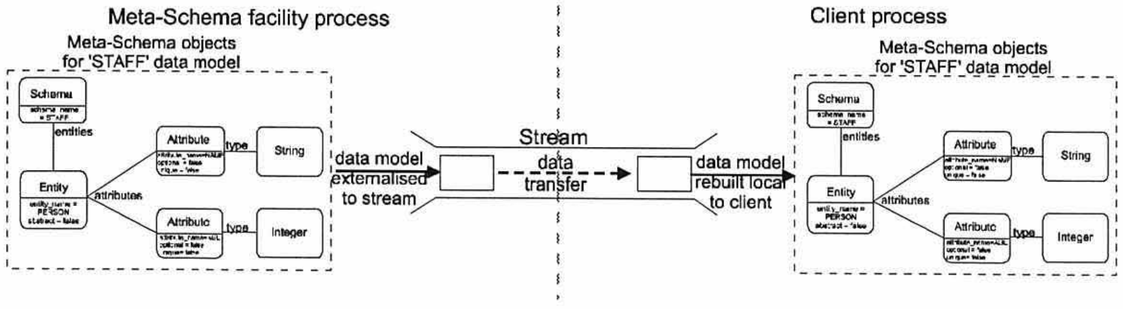

Meta-Schema facility process
Meta-Schema objects for 'STAFF' data model
Schema
entities
Entity
attributes
Attribute
type
String
Integer
data model externalised to stream
Stream data transfer
data model rebuilt local to client
Client process
Meta-Schema objects for 'STAFF' data model

Figure 7.14: *Meta-Schema Objects representing entity meta-information can be copied to client processes for fast access using a stream*

the local process. This is a similar situation to that of caching entity data objects local to client applications in the Persistent Data Access Service, to pass data by value rather than by reference which the CORBA model is based on. The solution is also similar to the use of streams to move the meta-information between processes and to rebuild the Meta-Schema objects in the client process(Figure 7.14).

The MetaSchemaFacility interface(Figure 7.15) provides the pass-by-value ability to clients wishing to copy data models locally. A client can request a stream containing data model meta-information using the MetaSchemaFacility interface. The stream's data can then be moved locally possibly using the Stream Tunnel Service. As with the Structured Entity Stream Format, the format that the meta-information is held in the stream is specified in the Meta-Schema Stream Format(Appendix E).

```
typedef string Identifier;
typedef sequence <Identifier> Identifiers;

interface MetaSchemaFacility : CosLifeCycle::LifeCycleObject
{
   Identifiers get_schema_list();
   Identifiers get_entity_list(in Identifier schema_name) raises(NotFound);
   Identifiers get_schema_defined_type_list(in Identifier schema_name)
   raises(NotFound);
   Identifiers get_schema_constant_list(in Identifier schema_name) raises(NotFound);

   CosStream::Stream
      get_schema_meta_info(in Identifier schema_name) raises(NotFound);

   CosStream::Stream
      get_entity_meta_info(in Identifier schema_name,
                           in Identifier entity_name) raises(NotFound);
   CosStream::Stream
        get_schema_defined_type_meta_info(in Identifier schema_name,
                                          in Identifier defined_type_name)
                                          raises(NotFound);
   CosStream::Stream
      get_schema_constant_meta_info(in Identifier schema_name,
                                          in Identifier constant_name)
                                          raises(NotFound);

};
```

Figure 7.15: *The MetaSchemaFacility interface can be used to transport meta-information using a stream to a client process*

7.3 Demonstration of the Operation of the Persistent Data Access Service

This section provides a brief synopsis demonstrating the operation of the Persistent Data Access Service(PDAS) and shows how the various components of PDAS work together to carry out the task of giving access to persistent data.

The scenario used for the demonstration is typical of a common programming procedure to read or update entity attributes. The steps common to such a procedure are:-

1. Open a session to connect to a datastore.
2. Start a transaction.
3. Execute a search for an initial entity.
4. Navigate relationships from the initial entity to obtain references to other entities that are of interest.
5. Read or update the attributes of entities.
6. Commit the transaction
7. Close the session.

The data model for this scenario is very simplistic, in which a 'well' entity has two attributes:- a name and a reference to a 'wellbore' entity. The 'wellbore' has a single attribute that records how deep the 'wellbore' is. The scenario will use PDAS to search for a 'well' by its name, navigate the 'wellbore' relationship and change the wellbore's 'depth' attribute. The scenario is described in the following steps:-

1. The first step is for the client to create a transaction-aware session. To do this, the client must firstly obtain a reference to a DatastoreServer object that can start a ServerSession to the required datastore. The client then uses the *create* operation of a SessionFactory to create a Session object.
2. In response to a *create* operation, the SessionFactory uses the DatastoreServer to start a ServerSession.
3. The ServerSession will create a datastore session to access it.
4. The SessionFactory also creates the client side Session. Upon creation the client side Session will grab a reference to the ServerSession's DataObjectServer and then create an EntityDObManager.
5. The EntityDObManager will use the Stream Tunnel Service to setup a stream between itself and the DataObjectServer.

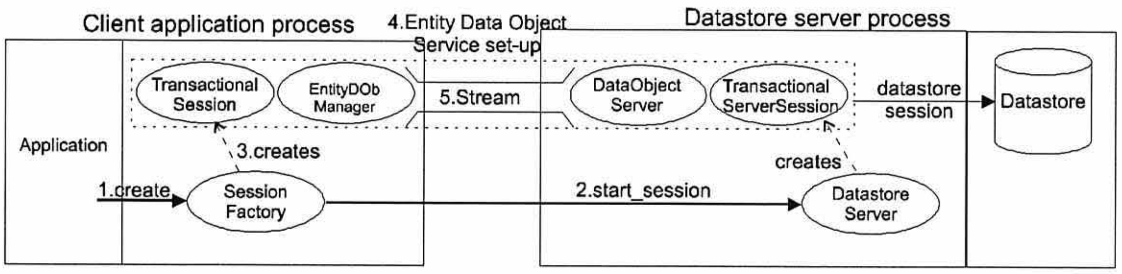


Figure 7.16: *Steps 1-5, creating client and server sessions and the setting up of the Entity Data Object Service and a stream*

6. The initialisation of the PDAS components and the session is complete. The situation is shown in figure 7.16.

7. Before carrying out any manipulation, a transaction must be started. The application must use the Object Transaction Service to create a transaction. As the transaction extensions of Session and the ServerSession are transaction aware, all manipulation of entities will be within the context of a transaction.
8. The next stage involves executing a search for the desired 'well' entity and retrieving it. To do this the application must first get the Session's QueryEvaluator. The QueryEvaluator is the Query service's interface that allows the execution of a search.
9. The application will use the query "select from well where name = 'Alpha00001'" to find the desired 'well'. The application executes this query using the Session's QueryEvaluator.
10. The Session's QueryEvaluator forwards the query to the server side QueryEvaluator.
11. The server QueryEvaluator translates the query into the datastore's native query language and executes it.
12. The result of the query is a set of oid's that are passed back to the client Session. In this case the result is only a single oid which is the desired 'well' (Figure 7.17).

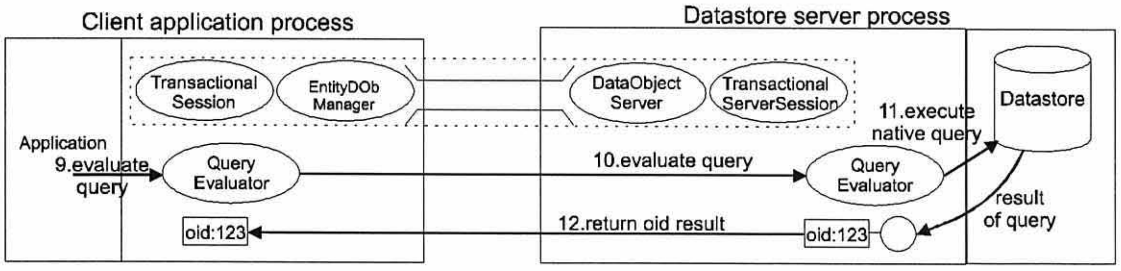


Figure 7.17: *Steps 9-12, executing a query and the returned result is an oid(s)*

13. The client QueryEvaluator uses the oid to set the Data Object identifier of the EntityDObManager and requests the manager to retrieve the entity.
14. The EntityDObManager responds by requesting the DataObjectServer to retrieve the entity.
15. The DataObjectServer adds the indicated entity to a EntityContainer.
16. The EntityDObManager pulls the EntityContainer from the stream.

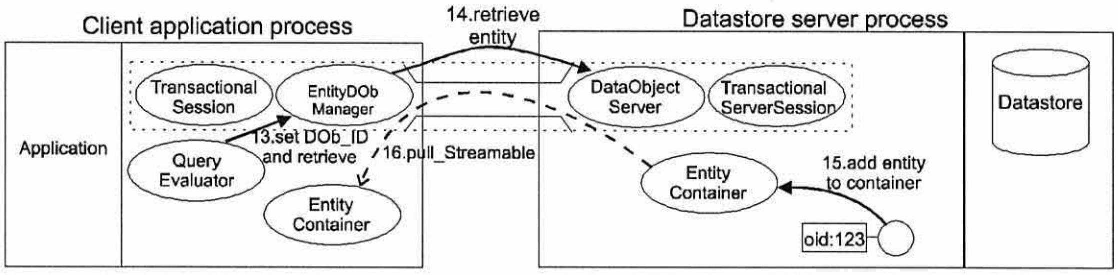


Figure 7.18: *Steps 13-16, the entity indicated by the oid is retrieved using the Entity Data Object Service. The entity is transported in a EntityContainer through a stream.*

17. The EntityDObManager extracts the entity from the EntityContainer and adds it to the client's entity cache.
18. An EntityReference to the entity is passed to the application as the result of the query.
19. The application requests the operation *get_Entity("wellbore")* on the EntityReference.
20. The relevant entity is not in the cache, so the EntityReference initiates the retrieval of the entity using the EntityDObManager. The identifier is a Entity_DOb_ID with attributes oid=123 & attr_name="wellbore".
21. The DataObjectServer retrieves the object from the store and the entity is transported through the stream to the client's cache(Figure 7.19).

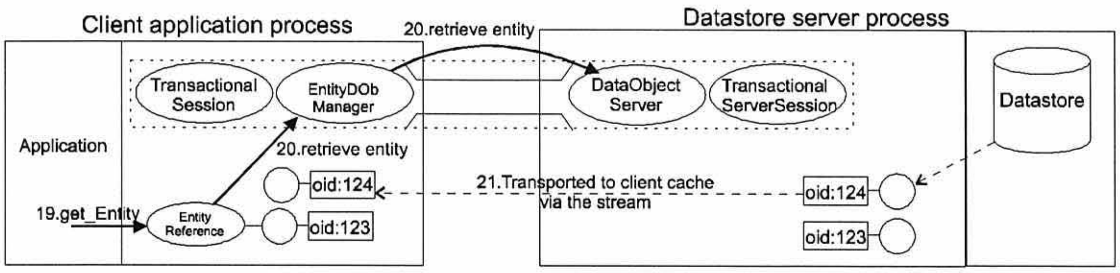


Figure 7.19: *Steps 19-21, the referenced 'wellbore' entity is retrieved using the Entity Data Object Service*

22. An EntityReference to the newly retrieved 'wellbore' entity is passed back to the application.

23. The application changes the depth by calling *set_Double("depth",378.56)* on the EntityReference.
24. The application has finished its manipulation, therefore it can include its updates in the transaction commit by calling *save_all_updates* on the Session. Calling this operation causes all entities that have been changed to be transported back to the ServerSession and the changes written to the datastore.

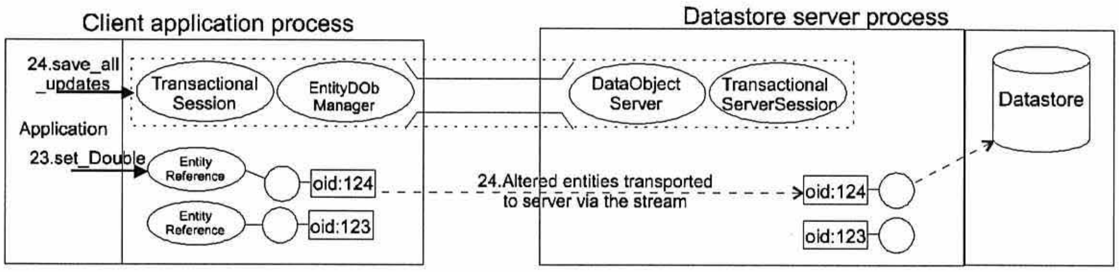


Figure 7.20: *Steps 23-24, the 'depth' attribute of the 'wellbore' is changed, all updates to cached entities are transported back to the server and written to the datastore*

25. The application must then commit the transaction. Any data that has been saved within the context of the Session with be included in the datastore's transaction commit.

7.4 Summary

This chapter has introduced the Persistent Data Access Service(PDAS). The PDAS enables the manipulation of highly structured entity data. The chapter firstly reacquainted the reader with the high level design features and concepts described in chapter 4, which PDAS is based on. These design features and concepts include: -

- The Meta-Schema model that provides a common data definition model that proprietary data models map their definition to.
- The Meta-Schema model has a direct IDL interface translation, therefore CORBA objects represent the meta-information contained in data models. This meta-information is crucial to the service to enable the dynamic management of entities.
- The use of sessions to access persistent data, represent a datastore connection and integrate with the transaction and concurrency services.
- Reuse of the Stream Tunnel Service to transport data and extend the Data Object Service to manage and access the data.

- The assignment of object identifiers(oids) to entities as an independent language/platform means of representing references.

Next, the three main PDAS modules and their interfaces were presented. These are the actual Persistent Data Access Service(PDAS) module, the PDAS Server module and the Entity Data Object Service

The PDAS module contains interfaces that are the application's access to the service. These interfaces include:-

- The Session interface for basic entity management, access to other facilities of the service and client side integration with the concurrency and transaction services.
- A SessionFactory for creating Sessions.
- An EntityReference interface which is used to manipulate the attributes of entities.

The PDAS Server module has a DatastoreServer interface for the creation of ServerSessions that concurrently access the supported datastore. The ServerSession integrates the server side with the transaction and concurrency services and provides a DataObjectServer for the retrieval and storing of entities from/to the datastore.

The Entity Data Object Service(EntityDObS) is an extension of the Data Object Service to enable the transport of entities between the client and server session components. It specifies the Data Object Identifiers for indicating the location an entity or entities, an EntityContainer for the holding of entities and an entity serialisation format that specifies how entities are serialised to a stream.

Access to meta-information described using the Meta-Schema model is important to many PDAS components to enable the dynamic handling of entities. However, if the Meta-Schema objects are remote to the client and the objects are frequently accessed, then performance of the system will dramatically decline. The MetaSchemaFacility provides a way for clients to obtain a local copy of the Meta-Schema objects.

The final section describes a typical scenario in which the PDAS might be used. The scenario included performing a search for an entity, retrieving the result of the search, navigating a relationship from the entity and retrieving the entity which is referenced.

In conclusion, the functionality provided by the PDAS is similar to the basic functionality provided by facilities such as POSC's Data Access and Exchange Facility, STEP's Standard Data Access Interface and other data manipulation facilities, in that they present interfaces to applications that allow the applications to manipulate persistent data. However, these facilities usually put many constraints on the data that can be manipulated such as the data definition language that may be used, or the language that the application is written in. PDAS does not have these constraints as data models defined in proprietary data definition languages can be mapped to the Meta-Schema model. Also the service is specified in IDL, and therefore is language-independent.

PDAS is a open specification for a set of interfaces with guidelines on how they interact. As with all CORBA specifications no implementation detail is specified. There are only guidelines on component behaviour and a definition of their interfaces. An especially powerful interaction is the ability to transport entities between the client and server sides of PDAS using the Stream Tunnel Service and the Structured Entity Stream Format. The combination of the mechanism to transport structured entity data and of the Meta-Schema model to describe the entity data makes clients entirely independent from the proprietary nature of datastores. This insulates client applications from the specifics of datastore types and allows the creation of generic clients that can connect to any datastore supporting the PDAS server side and manipulate its data. Also, if clients have no dependency on a specific datastore, then the application code is more portable and reusable across datastore types.

Chapter 8

Roles of the Persistent Data Access Service

8.1 Introduction

This chapter discusses various roles and factors involved in the roles that the Persistent Data Access Service(PDAS) can be used in, including the following:-

- Supporting arbitrary datastore types.
- Supporting the POSC Data Access & Exchange(DAE) interface and the Epicentre model and its influence on the design of PDAS.
- Using PDAS in conjunction with a database adaptor.
- Using PDAS as a standard data access interface for data transformation using the Mapping Manager Architecture.

8.2 Supporting Arbitrary Datastore Types

The purpose of the Persistent Data Access Service(PDAS) is to provide a way for applications to manipulate data resident in heterogeneous datastores independent of the proprietary datastore type storing the data(Figure 8.1). Thus, providing application insulation from the datastore's proprietary interface and software code means no need to link client applications with proprietary code library.

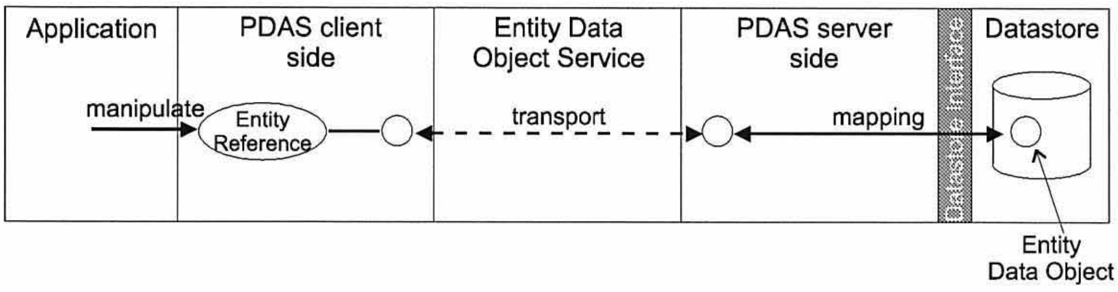


Figure 8.1: *The Persistent Data Access Service used to manipulate data in heterogeneous datastores*

This independence is partly due to the strongly specified Entity Data Object Service(EntityDObS). EntityDObS allows the retrieval and storage of structured entities in conjunction with the Stream Tunnel Service and the Structured Entity Stream Format for the transport of the serialised data. The client side of the service is therefore fairly straightforward; it has to manage the retrieved entities in the local process and hand out EntityReference objects allowing the manipulation of entities by applications. Consequently, the client side of the service is entirely generic as it can handle entities from any datastore type that is supported by the server side.

The real complexity in providing this independence comes from the mapping carried out by the server side of the service to transform entity data to/from its stored format and a format suitable for passing through/from EntityDObS. Each datastore type will have to have server side software that is specific to the datastore type to perform this mapping.

The mapping performed by a PDAS server is crucial to the service supporting arbitrary datastore types and providing independence on the client side from the datastore. There is a number of responsibilities involved in the mapping that a server side must perform to make data available to clients. These responsibilities include the following:-

- Generating the calls to the datastore's interface to retrieve and store entities.
- Transforming data between its native form used in the call to the datastore interface and a form that can be passed to/from EntityDObS. This will include transforming the data so that it reflects its description in the meta-schema model e.g. converting a native datastore data type to an entity to represent the type.
- Assigning oids to retrieved entities. PDAS requires entities to be identified by a unique session dependent integer allowing the independent representation of references to entities.

- Maintaining a list of handles to retrieved entities and the oids assigned to them.
- Logic is required so that when entities are stored, their persistent form in the datastore is consistently changed to mirror the changes made by the applications in PDAS.
- Providing functions to create and delete entities.
- Mapping transaction and locking acquisition operations of the transaction and concurrency services to the native datastore interface.

These duties are easier to implement for some datastore types than others. For example, an object-oriented database will already provide an oid facility. Also, the logic required to make changes to the persistent data consistent with changes made in PDAS is simpler for datastores that have an object-oriented nature such as OODBs and the DAE.

For datastores that do not have an object-oriented nature, such as relational databases, it is especially complex to be supported as a PDAS datastore. The major complexity comes from having no unique identity for entities/tuples, because identity is based on the values of key attributes. This leads to complexities in maintaining the oid to entity/tuple pairs and greatly increases the complexity of the logic required to consistently store entities.

The mapping process should be automated, so that a PDAS server specific to a datastore type can automate the mapping of any data model that the datastore can support. This automated mapping can be achieved by gaining meta-information on the entity types of the data model, including the type of each entity attribute. The meta-information allows the automatic generation of datastore retrieve and store calls. It also allows entities to be automatically converted to a standard form by converting each entity attribute to a corresponding attribute type in the standardised form.

The meta-information needed to perform the automated mapping is obtained from the meta-schema model description of a data model. A data model might not easily be described using the meta-schema model. For example, the meta-schema model might not support the data types of the data model incorporates, therefore they have to be mapped to an entity type or be accessed using a DescribedType(see chapter 4). Another example of a mapping difficulty is that relational databases do not explicitly define relationships as attributes of tuples, but they are only defined as foreign keys. The meta-schema model defines a relationship as an explicit entity reference attribute of an entity. Thus, to define a relationship between tuples in the meta-schema model, the entity representing the tuples has to have additional reference attributes representing the tuple's foreign keys. Complexities such as these have to be taken into account in the automated mapping process.

8.3 Supporting the POSC Data Access & Exchange Interface and the Epicentre model

A goal has been to support datastores that implement the POSC Data Access & Exchange(DAE) interface to access instance data of the Epicentre model. A PDAS server side supporting the DAE will have to satisfy the responsibilities presented in the previous section.

The aim to support the a DAE datastore has had a major influence in the design of the Persistent Data Access Service(PDAS) and its underlying services - DObS and STS. The features that influenced the design of the services are:-

- The nature of Epicentre data that ranges from complex networks of objects to multi-megabyte collections of scientific data.
- The sheer size of the Epicentre data model with 1500 objects/entities, each entity having many attributes and relationships, and a complex inheritance hierarchy.
- Epicentre has many specialised data types to represent various quantities, geographical locations and spatial structures.
- The DAE has no universal unique identifiers for entity instances, only session dependent handles to entity instances.
- Typically, applications will repeatedly traverse the same relationships from a selected root object. This can be optimised by specifying a template of the entities and their relationships to be traversed, therefore allowing a predetermined group of entities to be retrieved and stored in a single request.

PDAS can support the differing types of Epicentre data. It can efficiently handle the complex structured entities that Epicentre contains e.g. the 'well' entity(chapter 2). The Structured Entity Stream Format allows these complex entities to be serialised to and from a stream for transport. The meta-information provided by the meta-schema model allows components to understand an entity's structure to dynamically handle it.

For the massive collections of scientific data, the main consideration is the utilisation of an efficient transport mechanism to get the data from server to client and vice-versa. The Stream Tunnel Service has been proven to be an efficient mechanism for transporting such large amounts of data. This efficiency comes from using low-level network transport mechanisms and also has the possibility of using more efficient network transport mechanisms e.g. connectionless UDP sockets, with an especially tailored data consistency protocol.

The immense size of the Epicentre data model has had the most significant effect on the design of the Persistent Data Access Service. The size of the text file for Epicentre's EXPRESS description is over 600 kilobytes. The size of the compiled code containing static IDL code to handle each Epicentre entity would be many orders of magnitude larger than its text description and therefore simply unfeasible to link with applications. This factor drove the requirement for components to dynamically handle entities, thus the requirement for access to meta-information on a data model. The meta-schema model was designed to meet this need for meta-information on data models, and it also describes data models based in other disparate data definition models.

The meta-schema model has a small selection of basic data types, while Epicentre has an extensive collection of complex data types to represent quantities, geographical locations and spatial structures. These data types cannot be directly represented in the meta-schema model, but there are two ways they can be represented. The first way is to represent them as DefinedTypes. The meta-schema DefinedType is made up of a set of string identifiers indicating the type kind and an array of binary data storing the type's instance data. Applications manipulating DefinedTypes are expected to know the internal structure of the type's binary data. The second way is to map each data type to an entity and access the type's data through the attributes of the entity.

To manipulate an Epicentre entity using the DAE, a handle has to be obtained through the DAE to the entity. The handle is obtained by performing a search or traversing a relationship from another entity. A handle is a language specific structure and is only valid within the current session, hence is useless outside the session and process that obtained the handle. One of the requirements of PDAS is to cache data locally in client processes, thus a handle needs some sort of representation in the client process. This lead to the need for object identifiers(oids) to be associated with DAE handles, to represent the handle in the client process. The manipulation of the oid representing a handle to an entity in the client process is applied to the actual DAE handle upon storing of the entity.

Retrieving/storing a predetermined group of entities by specifying the relationships that should be traversed between the entities is an important optimisation factor. Usually an application will frequently retrieve and store the same set of entity types. By specifying a template of the entity types and their relationships so that the group of entities can stored/retrieved in a single request has major performance benefits.

This situation of retrieving/storing constant group of entity types was typical of the mapping of OpenSpirit components to Epicentre entities. A single OpenSpirit component maps to many Epicentre entities, where the mapping would be described in Expressive. The Data Object Retrieval Map(section 4.5.8) can be used to specify a template of the entities involved in a mapping of an OpenSpirit component, therefore improving the performance of the retrieval/storage of the Epicentre entities. PDAS allows a retrieval map to be specified in the retrieval of a relationship in the operation *get_Entity_using_map* of the EntityReference interface.

8.4 Using the Persistent Data Access Service as a Database Adaptor

The Persistent Data Access Service(PDAS) is a service for the explicit access and manipulation of persistent data. In contrast to this, a database adaptor(section 3.4.2) provides an implicit persistence mechanism to CORBA objects. The database adaptor takes care of data access and manipulation to its supported database and of activating CORBA objects when needed. The impact these differing forms of persistent data access have on applications is that for PDAS, the application has to explicitly access and manipulate data itself whereas with a database adaptor the application has no notion of a CORBA object and its attribute data are persistent.

However, the database used in conjunction with a database adaptor should ideally have a model similar to that of the object-oriented model that CORBA is based on. This makes mapping between database data items easier, but the most crucial reason for this affinity between models is the ability to uniquely and universally identify data items within a database. If a data item has a unique identity for the lifetime of the data item then a value representative of its identity can be encoded into CORBA object references. Consequently, not only are the CORBA objects persistent but their object references are also persistent. A unique identity is part of the object-oriented database model, which also has an object model close to the CORBA object model. For these reasons, object-oriented databases are most commonly supported by database adaptors.

The Persistent Data Access Service(PDAS) can also be used in the role of a database adaptor(Figure 8.2) and has the benefit that it can support datastores other than object-oriented databases(OODBs). Due to the mapping performed by the PDAS server side, all datastores appear as a datastore supporting the meta-schema model, which is object-oriented. However, it is not truly like the object-oriented database model due to PDAS oids are only unique within a session and not universally unique for the lifetime of an entity as in OODBs.

The additional functionality that has be provided by a PDAS database adaptor is the following:-

- The binding-together of CORBA objects having a static IDL interface and entity instances

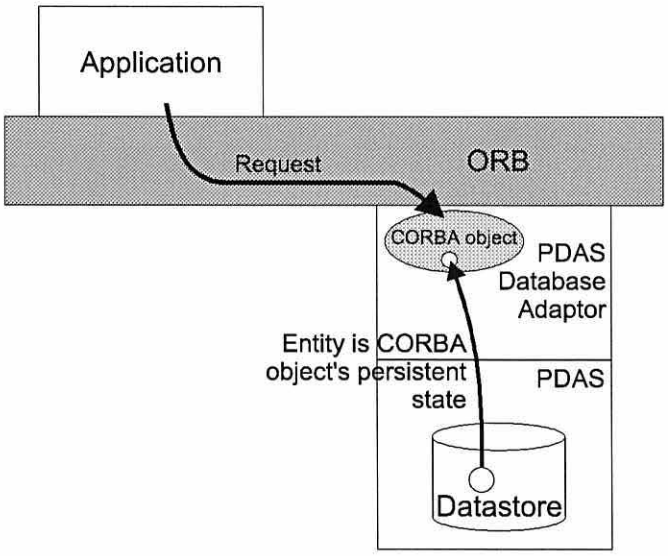


Figure 8.2: *A PDAS Database Adaptor allows an entity to be the persistent data of an CORBA object*

retrieved from a datastore.

- The embedding of session-dependent oids into CORBA object references.
- Activation and deactivation of CORBA objects bound to entities.

The binding-together of CORBA objects and entity instances is simply a matter of retrieving the referenced entity and creating a CORBA object that can manipulate the data of the referenced entity type. This process was described in section 4.5.9 and it entails a FactoryFinder being used to find a Factory object capable of creating a suitable CORBA object to bind to the entity.

An oid to an entity is obtained from the entity's EntityReference object. Once obtained, the oid can be embedded into a CORBA object reference and the reference can be made known to the CORBA environment.

Activation of CORBA objects happens when a request is received for a CORBA object that is not instantiated. To activate the CORBA object, the embedded oid will be extracted from the object reference and the entity referenced by the oid is searched for using the *find_entity_by_oid* operation of the Session interface. An EntityReference object is returned if the referenced entity is found. If the entity is found then the adaptor activates the CORBA object by creating and binding a CORBA object to the entity.

Deactivation is the destruction of active CORBA objects according to some policy such as not serving a request for a certain period of time. The deactivation deletes the CORBA object instance, but it does not release the entity retrieved using PDAS. This allows any CORBA object references to the entity retained by components in the CORBA environment to still be valid. If the reference is used after deactivation then the entity is reactivated with a new CORBA object shell. This differs from OODB adaptors that will totally deactivate the whole CORBA object and its persistent object. Here, the entity has to remain active due to its association with its oid.

Using a database adaptor in conjunction with the Persistent Data Access Service relieves applications of the responsibility to explicitly access and manipulate the persistent data. But the application must be aware that a CORBA object reference to an object activated in a PDAS database adaptor is only valid for the duration of the current session.

8.5 Using PDAS as a Standard Data Access Interface for the Mapping Manager Architecture

The Mapping Manager Architecture(MMA)(see section 3.2.2) is a methodology for moving and transforming data between different data models and datastores. The principal factor in the MMA needed to achieve its purpose is the Expressive language. The Expressive language allows the formal definition of the data transformation between two models. Given an Expressive definition, code has to be produced that retrieves the data from a source datastore, transforms the data and stores it in the destination datastore. The problem with this is that each datastore type involved in the mapping will have a different data access interface. Thus, if this code generation was automated, a code generation tool would have to be written for each datastore type needed to be accessed(Figure 8.3). Writing such a tool for just a single datastore type is a highly complex undertaking and the resources needed to write tools for many datastore

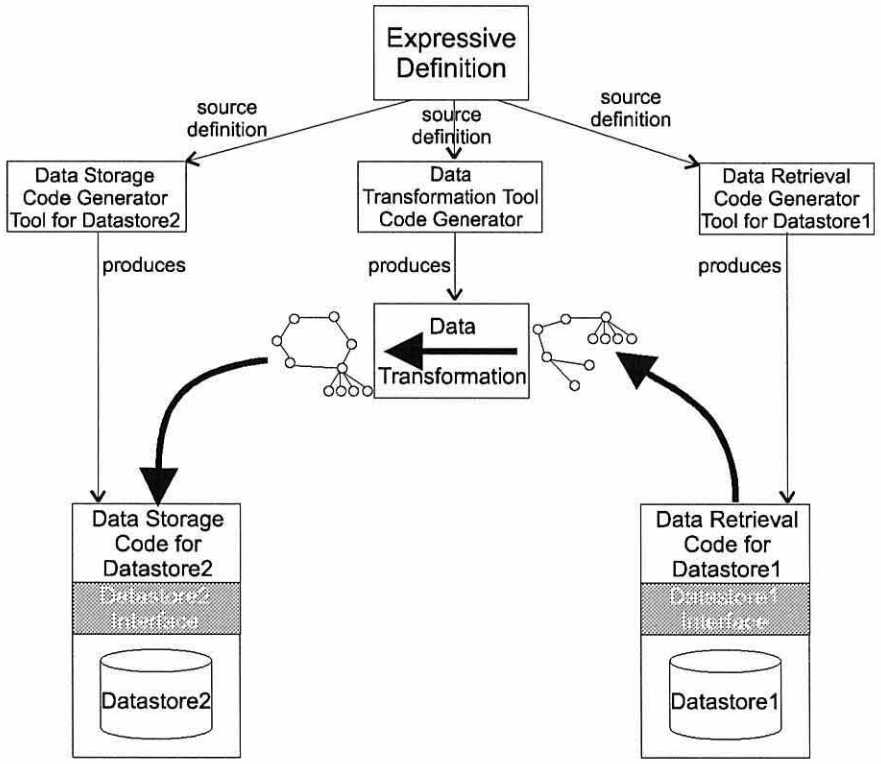

Expressive Definition
source definition
source definition
source definition
Data Storage Code Generator Tool for Datastore2
Data Transformation Tool Code Generator
Data Retrieval Code Generator Tool for Datastore1
produces
produces
produces
Data Transformation
Data Storage Code for Datastore2
Datastore2 Interface
Datastore2
Data Retrieval Code for Datastore1
Datastore1 Interface
Datastore1

Figure 8.3: *Generation of code to automate the transformation of data in the MMA, each datastore type needs a code generation tool specific to its proprietary data access interface*

types would be impractical.

The Persistent Data Access Service(PDAS) provides a layer that makes all datastores seem to have the same data access interface and data definition model. This greatly simplifies the task of retrieving and storing data being used in the data transformation(Figure 8.4). The only code generation tools needed would be for the actual data transformation and to retrieve/store data through PDAS. Therefore, this makes the need for the many code generation tools for different datastore types redundant. The use of PDAS removes from the MMA the need to incorporate the complexity of proprietary data access interfaces, so that the MMA can concentrate on what it is aimed at, which is data transformation.

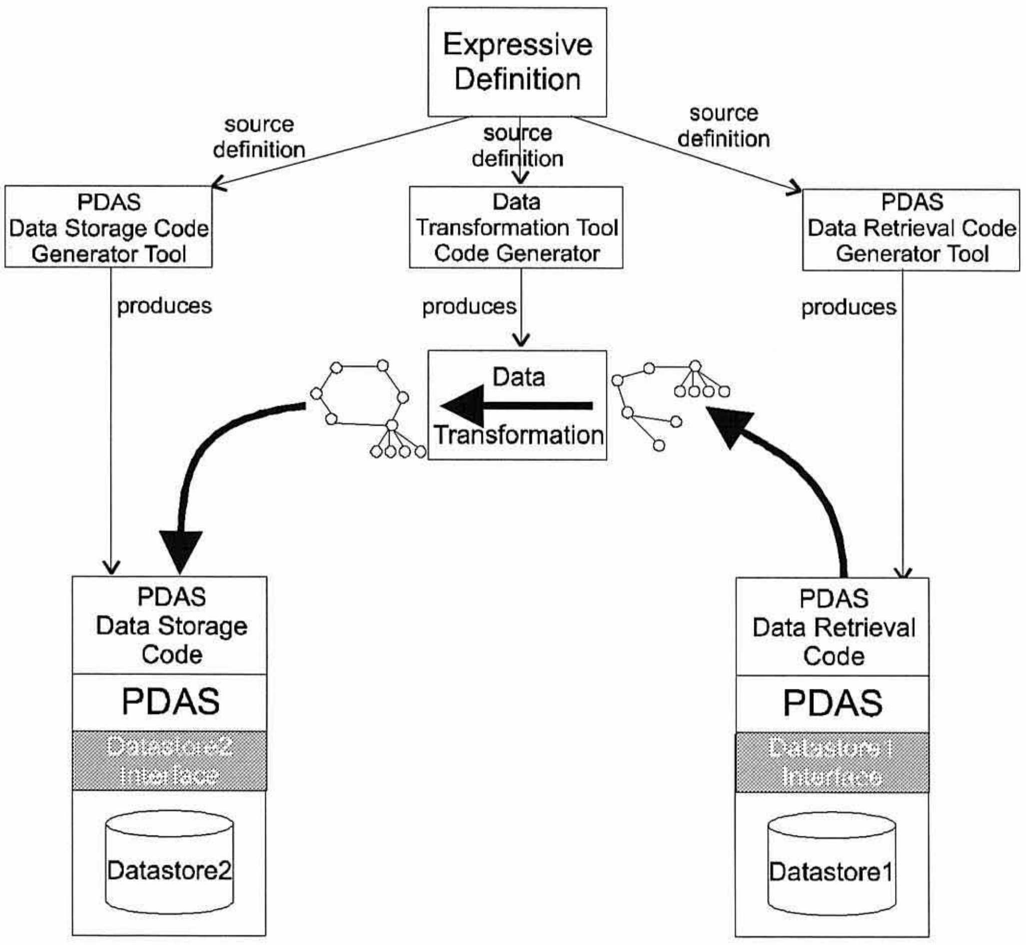


Figure 8.4: *PDAS is used to insulate the MMA from the proprietary data access interfaces of the individual datastore types, the other code generation tools needed are for accessing the PDAS interface and for transforming the code.*

8.6 Summary

This chapter has discussed some of the roles that the Persistent Data Access Service(PDAS) could perform, including supporting access to arbitrary datastore types and more specifically the POSC DAE and the Epicentre model. It also indicates how PDAS can be used as a database adaptor and how it would be of great value to the Mapping Manager Architecture.

In supporting arbitrary datastores, the datastore specific server side of PDAS has to perform a number of duties in making its stored data available to PDAS clients. These responsibilities include data fetching/storing/transforming, object identifier(oid) management and transaction/concurrency management.

Supporting a DAE datastore also requires the PDAS server side to perform these tasks. Considering how DAE and Epicentre could be supported has driven many design decisions of PDAS. The variety of Epicentre data, such as large arrays of data and complex networks of objects, drove the need for the serialising of multiple objects into a stream and for use of efficient stream transport mechanisms. The immense size of the Epicentre model requires the dynamic handling of data since compiling static handling code for a model of this size is unfeasible. DAE entity handles are confined to the process space of the DAE session. This drove the need for a representation of the handle outside the process space, thus the association of oids with entity handles.

The database adaptors enable implicit persistence of CORBA objects. PDAS can be used in such a way by adding binding of CORBA objects to entities, embedding entity oids into object references and enabling CORBA objects to be (de)activated. However, the drawback is that CORBA object references referring to objects created by the PDAS database adaptor are only valid within the session that created them i.e. they are not valid between session instances.

PDAS makes datastores appear to have the same data access interface and data definition model. The Mapping Management Architecture requires data retrieval and storage from/to many types of datastore. The common data access interface that PDAS provides allows the MMA to avoid proprietary interfaces to datastores.

Chapter 9

Conclusions

9.1 General Overview

The work presented in this thesis is an investigation into persistent data access in the CORBA environment. This has lead to the designing of a set of CORBA services to provide a standard means of accessing persistent data stored in heterogeneous datastores.

The need for such a service was first suggested in chapter 2. This chapter provides introductory material on the CORBA architecture and the CORBA services, as well as an overview of the STEP and POSC SIP architectures. These architectures provide inter-operability for applications, but in different ways. CORBA provides dynamic inter-operability, where components directly request services of each other. STEP and SIP provide inter-operability by applications sharing persistent data that have a defined data model. Persistent data is critical to many applications. STEP & SIP strongly support access and manipulation of persistent data, but CORBA is very weak in this area.

Chapter 3 examines current methods that are being used to access persistent data in CORBA. These methods can be divided into three categories:- proprietary solutions, standard IDL interface solutions and persistent CORBA objects using database adaptors.

Proprietary data access solutions are the most prolific method in use with CORBA systems. Here, CORBA components directly access persistent data using the datastore's proprietary interface. There are many problems associated with using datastore proprietary interfaces, including dependency on the datastore's interface, work required from application developers to cross the impedance mismatch, poor reusability and flexibility of code. OpenSpirit's data access approach is a typical example of these problems.

Currently, there are only two standards to provide CORBA applications with access to persistent data. These are OMG's Persistent Object Service(POS) and STEP's IDL Standard Data Access Interface(SDAI). POS should have been CORBA's default method of accessing persistent data, but as chapter 3 extensively indicates, there are many design faults and ambiguities in POS. The OMG has retired POS in response to the discovery of these problems. The failure of POS has lead developers to implement their own proprietary data access solutions. The IDL binding of SDAI is STEP's answer to accessing data through CORBA. However, IDL SDAI is highly embedded in STEP concepts and technologies.

Database adaptors allow CORBA objects to be persistent. Thus, data access is indirectly achieved through a CORBA object interface. Database adaptors fit in well with the CORBA object model and have the benefit that applications are totally unaware that CORBA objects are persistent, thus have no code dependency with the adaptor or its supported database. Database adaptors depend on the supported database having a unique identity value for stored data items. This value can be embedded in CORBA object references to represent internal relationships in the external CORBA environment. However, many data definition models(e.g. relational) and data access interfaces(e.g. DAE) do not support unique identifiers, thus they cannot be used with database adaptors.

Chapter 4 summarises the problems with CORBA persistent data access solutions and puts forward a set of requirements for a solution for these problems. The most prominent of these requirements are:- a set of CORBA services to access data in heterogeneous datastores, fit in with existing data access legacy applications, efficient manipulation of data either locally(cache data) or remotely, as well as integration with the CORBA transaction & concurrency services.

The chapter continues by presenting high-level design decisions for a Persistent Data Access Service to achieve these requirements. A compendium of these design decisions is:- the use of meta-information to dynamically handle and map data objects, the Meta-Schema model as a standard data definition model to represent data models from heterogeneous datastores, a stream mechanism to cache data, session components to represent a connection with a datastore and integrate with transaction & concurrency services, as well as associating object identifiers with active data objects to represent relationships externally to the datastore.

A design decision was for the need for a stream mechanism to allow the transport of non-IDL defined data for caching data purposes. This is realised in the form of the Stream Tunnel Service(STS) described in chapter 5. STS provides a set of interfaces to permit the setup and use of distributed streams. The actual mechanism to transfer data does not necessarily have to be an ORB, since low-level network data transport mechanisms can be employed to transfer data for performance enhancement purposes. This enhancement was proved in an experiment comparing an ORB-based stream and a socket-based stream. The experiment showed socket based streams become more efficient when sending blocks of data over 6Mb. STS could be useful in other areas such as:- providing a CORBA copy-by-value facility, multimedia data transmission with UDP sockets, event broadcasting and encrypted communication channels.

The Stream Tunnel Service provides an excellent data transfer mechanism, but in respect to data access it has no means of managing data i.e. identifying, creating/deleting and retrieval/storage of data. The Data Object Service(DObS) described in chapter 6 is designed to provide these data management functions, as well as strongly integrating STS into it to provide data transport. DObS is only an abstract service, and must be extended to provide actual data identification and manipulation operations for the particular type of data being managed. The chapter goes on to present the File Data Object Service(FileDObS), which is an extension of DObS to permit access to files and directories. An implementation of FileDObS is described, along with a Java application that uses FileDObS to view/manipulate certain file types.

DObS has many similarities to that of the failed Persistent Object Service. The main differences between the two is that DObS offers a data access approach to accessing persistent data, where as POS concentrates on providing persistence to CORBA objects and this makes DObS more flexible. But the crucial difference that makes DObS work and POS not work is DObS's integration of a well designed efficient data transport service i.e. STS.

During the course of implementing these CORBA services, much knowledge has been gained and many lessons have been learnt. This knowledge can be described by a set of guidelines(Section 9.3). An example is the careful selection of system software i.e. the ORB product, by weighing up the need for code portability against the need for specialised features that are proprietary to a certain product. Another guideline was that all components of an application should keep to inter-operating through IDL interfaces, as this maximises flexibility and scalability. An area where keeping to this rule is difficult is in creating CORBA objects due to the need for language specific creation constructs. This difficulty can be minimised by making careful use of the Factory and FactoryFinder objects of the LifeCycle service.

Chapter 7 presents the final solution to the problem set, which is the Persistent Data Access Service(PDAS). PDAS is designed to allow access and manipulation of complex entity data, where the structure of each entity (its attributes, relationships and inheritance hierarchy), are described in the Meta-Schema model. PDAS also provides the ability to manipulate entities locally by caching the data in the client process.

PDAS consists of three modules:-

- The PDAS module that contains interfaces used by client applications to access the service's functionality and manipulate entities.
- The Entity Data Object Service for transport and management of entity data.
- The PDAS Server responsible for server session creation and data mapping.

Components implementing these interfaces and having the behaviour discussed in chapter 4 provide a complete open solution to accessing and manipulating persistent objects in the CORBA environment.

PDAS can be used in a number of different roles. Chapter 8 discusses these roles and the factors involved in their implementation. The chapter discusses using PDAS for its primary role, that is providing insulation to client applications from proprietary datastore types. The critical element to providing this insulation is the mapping the server side performs to make data reflect its Meta-Schema description.

A major influence on the design of PDAS was the need to support POSC's DAE and Epicentre data. These influences and their effect on PDAS's design are outlined, including the influence of the nature of Epicentre data, the huge size of the Epicentre model and the need to provide a representation of entity handles external to the session.

PDAS can also be used in the role of a database adaptor. To do this, extra functionality has to be built on top of PDAS, including binding of CORBA object to entities, embedding entity oids in object references and (de)activation of CORBA objects. However, the problem with this is that object references are not truly persistent, but are only valid for the duration of the session that created them.

The Mapping Manager Architecture is a methodology for moving and transforming data between data models and datastores. The great difficulty with performing such a task is writing the tools to automatically generate code to retrieve and store data to/from the necessary datastore types. Each datastore type will have a different data access interface, making the writing of the code generation tools a very resource consuming task. PDAS simplifies the task by making each datastore appear to have the same data definition model and data access interface, therefore needing for a single tool that produces PDAS retrieval/storage code.

9.2 Results of the Work

In the process of designing and implementing the CORBA services presented in this thesis, the following summarises the results of the work:-

1. *Realisation of the need for a CORBA service to access persistent data.* During the analysis of current CORBA persistent data access mechanisms, it became apparent that there was a important need for a CORBA service to access persistent data. This is due to the failure of POS to meet this need and the resulting widespread use of proprietary solutions to meet

developers needs. Database adaptors can satisfy this need but only for certain types of datastores.

2. *A set of requirements that a persistent data access service should satisfy.* As a result of the analysis of CORBA persistent data access mechanisms, a list of requirements was devised that a persistent data access service should meet the needs of.

3. *A high-level design of a service to satisfy the requirements.* A description of how the service should be structured to meet its requirements. The high-level design put forward many concepts necessary for the service to work such as the need for meta-information, streams, object identifiers and sessions. These design decisions were greatly influenced by the need to support POSC's DAE and Epicentre model.

4. *The Meta-Schema model.* The service is required to manipulate data resident in heterogeneous datastore types where these datastore types will have their own data definition models and languages. The Meta-Schema model provides a standard data definition model for the representation of the structure of data i.e. meta-information. The data definition models of supported datastores can be mapped to the model. The Meta-Schema model is designed to be as generic as possible to permit the mapping of various data definition models to it. In support of this the Meta-Schema caters for data definition models that use direct and attribute key based references, and the ability to support proprietary data types.

5. *The Stream Tunnel Service.* A specification for the set-up and use of distributed streams. STS is extremely useful to the CORBA architecture as it provides copy-by-value functionality, which currently does not exist in CORBA.

6. *Encapsulation of low-level network data transport mechanisms.* The StreamChannel interface of STS encapsulates the mechanism used to transfer data. Thus, low-level network data transport mechanisms can be employed for performance enhancement purposes. An experiment that was carried out proved this.

7. *The Data Object Service.* A specification for the management of persistent data including identification, creation/deletion and retrieval/storage of persistent data. This is an abstract service that enforces how these management operations are carried out and how it is integrated with the Stream Tunnel Service to transport data.

8. *The File Data Object Service.* A specification that extends DObS for the management and manipulation of files and directories. An implementation of the service was carried out with a Java client side and C++ server side. The service provides a network file system for Java applets that run in a secure environment preventing local file access. A sample application was implemented on top of the Java client side that permitted the viewing and manipulation of certain file types.
9. *The Persistent Data Access Service.* The definition of a set of interfaces to provide access and manipulation of complex entities resident in heterogeneous datastores, as well as the responsibilities of components implementing the service's interfaces.
10. *The Entity Data Object Service.* A service extending DObS for the transport and management of complex entity data. This includes the Structured Entity Stream Format that defines how entities are serialised to and from a stream. This service provides a mechanism for caching data local to clients.
11. *Integration with Concurrency and Transaction Services.* The Persistent Data Access Service has shown how integration with the OMG Concurrency Control Service and Object Transaction Service can be achieved.

The designed services including their IDL definitions, concepts and mechanisms have satisfied the aims of the work(section 1.5) together with the requirements defined in chapter 4.

9.3 Recommendations

The experience gained from researching, implementing and designing the services presented in this thesis has lead to the following suggestions for the design and implementation of future component software systems.

- Separate layers of software. Ideally, layers should use the three-tier architecture of applications, business objects and data storage. This divorces applications from business logic, and business logic from data storage, thus enhancing flexibility of layers.
- Add a persistence layer. In the three-tier architecture business objects, are highly dependent on the data storage mechanism in use. By adding a persistence layer that that encapsulates

the data storage mechanism, the dependency on the data storage mechanism is lowered. The Persistent Data Access Service is an example of a persistence layer in the form of a CORBA service.

- The easiest solution for persistent data is a database adaptor supporting persistent CORBA objects. Ideally, a data model should be structured to work with databases that have database adaptors e.g. object-oriented databases and relational databases with a relational to object-oriented mapping layer. However, database adaptors cause a very high dependency between business objects and their data storage mechanism. Also, they are difficult to use with existing legacy data and datastores.
- Careful selection of system software e.g. an ORB, should be made before any development takes place. The consideration should take into account the need for portability of code against the need for specialised facilities that proprietary to a specific product. Not only should the right product be considered, but which of the component technologies should the product be based on e.g. CORBA, Enterprise Java Beans and COM/DCOM. One of these component technologies might be better for the application needs than others.
- Systems that need to be reliable usually make use of transactions. For a CORBA-based system, the Object Transaction Service should be used for this facility. How the components of the system integrate OTS transactions should be investigated and designed before the start of development.
- The use of Java as the primary programming language of components has many advantages. The most prominent advantage is Java's portability characteristic of write once, run anywhere. This releases software from the confines of the platform it was written and compiled on. Java also provides a richly featured standard environment to write applications in, with the provision of the JDK class library that also contains the AWT. In relation to CORBA, Java's CORBA binding is effortless to use. This is due to Java's garbage collection i.e. no need to explicitly release reserved memory. Also there are no complexities related to pointers as with the C++ binding.
- Components should always inter-operate through IDL interfaces to preserve portability and flexibility. If components do not maintain this rule, then they are very difficult to separate as

they have a high dependency. Also, the CosLifeCycle service model of creating objects using Factory and FactoryFinder objects should also be used in support of this rule.

- Components needing to make frequent repetitive requests of another component should be located within the same process for fast local process request calls rather than slow inter-process requests. The Persistent Data Access Service took this into account with its requirement to cache data local to the client.

9.4 Future work

The Persistent Data Access Service(PDAS) only exists in a specification form, therefore future development work will consist of implementing the service. Modules of the service that are already implemented are the Stream Tunnel Service, the Entity Data Object Service, the Meta-Schema facility and a viewer of Meta-Schema data models. The Epicentre model has been mapped to the Meta-Schema model. The majority of complexity for the implementation will be involved in implementing the mapping of data from the datastore's stored form to its Meta-Schema description.

The implementation of PDAS will have the following stages:-

- Implementation of a Data Object Server for a relational database.
- Implementation of a Java generic client side of the service.
- Implementation of a Data Object Server for POSC's DAE datastore and Epicentre model.

Additional work will take the following form:-

- Implementation of a PDAS database adaptor.
- A tool to automatically produce CORBA objects that act as a static IDL interface to entities. These CORBA objects will be used in conjunction with the database adaptor.
- Using the Extensible Markup Language(XML) instead of the Structured Entity Stream Format and the Meta-Schema model to serialise and describe data.
- A tool to automatically produce code to retrieve and store data to/from a PDAS supported datastore described in an Expressive data transformation description.
- Disseminating the result and approach to standards bodies, such as the OMG and other practitioners in the field.

Appendix A

Glossary of Terms

API, Application Programming Interface. The definition of a set of functions.

Bauhaus principle. The principle that encourages the reuse of functionality that already exists.

Cache. The ability to store data in an area for fast accessing.

CAD, Computer Aided Design.

CCS, Concurrency Control Service. The CORBA service responsible for locking.

Client-server model. The architectural model where clients directly access databases and application servers.

Component. A piece of code that has a public defined interface and known functionality.

COM, Component Object Model. Microsoft's component architecture.

COOL-ORB. An implementation of an ORB from Chorus Systems.

Copy-by-value. The ability to pass an actual copy of a piece of data, rather than passing a reference to it.

CORBA, Common Object Request Broker Architecture. The OMG's specification for heterogeneous platform and language inter-operability.

CORBAFileView. The application demonstrating the File Data Object Service for distributed access to files.

CORBA object. A piece of active code/object fulfilling an IDL interface.

CORBA object reference. A handle to a CORBA object, that can be used to invoke operations of the object.

CORBA services. A set of services defined by the OMG to provide low-level functionality to CORBA components.

Data Access & Exchange. POSC's API specification for the manipulation of Epicentre model.

Data access interface. The API used to access and manipulate data stored in a datastore.

Database. A mechanism for storing data in a structured form.

Database adaptor. A specialised object adaptor providing persistence to CORBA objects.

Data Definition Language. The code used to describe the structure of data.

Data Definition Model. The model of how data is structured in a data model.

Data model. A description of the structure of data.

Data Object. An item of persistent data.

DataObjectManager. The interface for the management of persistent data in the Data Object Service.

Data Object Retrieval Map. The model that can be used to specify a group of data objects that can be automatically retrieved or stored in a single operation.

DataObjectServer. The interface that serves data to clients from its supported datastore

Data Object Service. The abstract service specification for the management of persistent data.

Data Storage Mechanism. The specific datastore product being used to store data.

Datastore. A mechanism for storing data in a structured or unstructured form.

DBMS, DataBase Management System. The software component providing a database's functionality e.g. querying, access control.

DOb_ID, Data Object Identifier. The interface used to specify the location of persistent data in the Data Object Service.

EJB, Enterprise Java Beans. Java's component architecture.

Entity. A unit of data that has a set of attributes, both referential and non-referential, and a inheritance hierarchy. An entity is the unit of manipulation of PDAS.

EntityContainer. An PDAS interface that allows the adding and removing of entities to/from the container. Once in the container, the contained entities can be transported through the StreamTunnel.

EntityDObS, Entity Data Object Service. The service extending DObS to transport structured entities in a stream. PDAS uses it to transport entities between client and server sessions.

EntityReference. The PDAS interface that allows the access and manipulation of an entities' attributes.

Epicentre. POSC's data model defining the structure of data items for the majority of objects that will be need to be stored in E&P information systems.

E&P, Exploration & Production. The sector of the oil & gas industry concerned searching for and processing of natural energy substances.

Express. The information modelling language for the modelling of complex schemata, including the definition of constraints amongst entities. Express is the language used to define Epicentre.

Expressive. The language developed by PrismTech to formalise data transformation.

Externalisation. The process of an object serialising its state data to a stream.

Factory. The LifeCycle service's interface that serves the purpose of creating CORBA objects.

FactoryFinder. The LifeCycle service's interface that can be used to search for a Factory object that is capable of creating an object of a specific type.

FileDObS, File Data Object Service. The service extending DObS to allow access and manipulation of files and directories.

GIOP, General Inter-ORB Protocol. The specification of messages enabling different ORBs to inter-operate.

IDL, Interface Definition Language. The OMG's language for defining the interface to CORBA objects.

IDL SDAI. The IDL binding to STEP's Standard Data Access Interface.

IIOP, Internet Inter-ORB Protocol. The specification for the exchange of GIOP messages over a TCP/IP communication link.

Internalisation. The process of an object reading its state data from a serialised form held in a stream.

Java. Sun Microsystems' language that is portable across platforms that have a Java Virtual Machine.

JavaIDL. Sun Microsystems' ORB that can used with JDK 2.

JNI, Java Native Interface. The mechanism used to execute platform specific code from Java.

LifeCycle Service. The service responsible for management of CORBA objects.

Meta-information. Information that describes the structure of an object/entity including attributes and inheritance hierarchy.

Meta-Schema Model. A model that allows the representation of meta-information, which describes data models of proprietary data definition models.

Meta-Schema Facility. An implementation of the Meta-Schema Model.

MMA, Mapping Manager Architecture. PrismTech's methodology for the transformation of data between models.

Multi-threaded. The ability of a program/application to have parallel execution of code within the same process.

Network Data Transport Mechanism. A mechanism that can be used to transport data over a network e.g. sockets.

NIST, National Institute for Standards and Technology. The standard's organisation that is the creators of STEP and Express.

Object adaptor. The element of CORBA that allows server objects to receive requests from the ORB.

Object-oriented. The paradigm that separates a problem into distinct entities, that have state and behaviour.

Octet Sequence. The IDL name for an array of bytes.

Oid, Object identifier. A value representing the identity of an object.

OMA, Object Management Architecture. The OMG's specification of how CORBA systems should be structures including common services, common facilities and applications.

OMG, Object Management Group. The organisation that created CORBA and the CORBA services.

OODB, Object-Oriented DataBase. A DBMS supporting the storage of data in an object-oriented form.

OpenSpirit. The PrismTech led alliance for the development of a 3-tier architecture for the integration of E&P datastores.

ORB, Object Request Broker. The layer of software that encapsulates distributed communication for CORBA applications.

OSEP, OpenSpirit Exploration & Production. The name given to the components defined in the OpenSpirit architecture.

OTS, Object Transaction Service. The CORBA service providing a distributed transaction facility.

PDAS, Persistent Data Access Service. The service providing access to persistent data resident in heterogeneous datastores.

PDS, Persistent Data Service. The POS interface that the implementation of serves data to persistent objects to/from its supported datastore.

PDASServer. The server module of PDAS providing interfaces, which the implementation of is specific to the datastore type being supported.

Persistent data. Data that exist beyond the life-time of the application that created it. The data is usually stored in some type of datastore.

PID, Persistent IDentifier. The POS interface to identify the location of data.

PO, Persistent Object. The POS interface supported by objects that are persistent.

POM, Persistent Object Manager. The POS interface that the implementation of provides request routing to an appropriate datastore.

POS, Persistent Object Service. The failed OMG service for accessing persistent data in heterogeneous datastores.

POSC, Petrotechnical Open Software Corporation. The E&P industry own standard's organisation.

Process. The execution environment of a running program.

PSS, Persistent State Service. The future replacement for POS. PSS will work in a database adaptor style to provide persistence to CORBA objects.

QueryEvaluator. The interface of the query service that permits the execution of queries.

Request. A distributed method call through an ORB to the interface of a CORBA object.

Relational database. A DBMS that stores data in the form of tables and relationships are represented in the form of keys.

Schema. A data model describing the structure of data.

SDAI, Standard Data Access Interface. STEP's specification for an API to access data defined in Express.

Semaphore. A facility preventing the interference of multiple parallel executing threads/processes on a shared resource.

SESF, Standard Entity Stream Format. The Entity Data Object Serivce's format which entities are stored in a stream in.

Session. A logical connection with a datastore. Also, the PDAS session interface gives clients access to the service's functionality.

SIP, Software Integration Platform. A set of POSC specification for the standardisation of information systems in the E&P industry.

Skeleton code. The code performing de-marshalling of request data for a CORBA object.

Socket. A low-level data transport mechanism for transporting data between machines on a network.

SSDF, Standard Stream Data Format. The Externalization service's format which data is stored in a stream in.

Stream. The Externalization service's interface representing a storage area for objects' serialised data.

StreamChannel. The STS interface representing one end of a distributed stream.

StreamChannelServer. The STS interface representing the slave end of a distributed stream.

StreamIO. The Externalization service's interface for read and writing data to/from a stream.

Streamable. The Externalization service's interface that is supported by CORBA objects that are capable of being externalised/internalised to/from a stream.

StreamTunnel. The STS interface for managing creation/deletion of streams and for pushing/pulling a Streamable through a stream.

STEP, STandard for the Exchange of Product Model Data. The name given to NIST's international standard for the exchange of product data.

STS, Stream Tunnel Service. The service for the set-up and use of distributed streams.

Stub code. The code performing marshalling of data for a request call.

TCP/IP, Transport Control Protocol/Internet Protocol. The network protocol of the internet.

Three tier architecture. The model of separating software into layers of :- applications, business objects and data storage.

TransferChannelCriteria. A data type allowing the definition of parameters for the set-up of a stream using STS.

Transaction. Operations performed within a transaction will have ACID properties of Atomicity, Consistency, Integrity and Durability. Transactions are critical to reliable systems.

Tuple. A row in a relational database table. Can be the equivalent of an object, where each column is an attribute.

UDP, User Datagram Protocol. A unconnected low-level network data transport protocol, for the fast transport of data between machines, although reliably.

XML, eXtendible Markup Language. A data format for structured document interchange.

Appendix B

Relevant CORBA Services

Presented here are the OMG services that are relevant to the design of the services in this thesis.

These services are:-

- Concurrency Control Service
- Externalization Service
- LifeCycle Service
- Object Transaction Service
- Persistent Object Service

B.1 Concurrency Control Service

```
// Concurrency Control  Service v1.0 described in
// CORBAservices: Common Object Services Specification, chapter 7

// OMG IDL for ConcurrencyControl Module, p 7-8

#include <CosTransactions.idl>
module CosConcurrencyControl {

                enum lock_mode {
                    read,
                    write,
                    upgrade,
                    intention_read,
                    intention_write
```

```
};

exception LockNotHeld{};

interface LockCoordinator
{
         void drop_locks();
};

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

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

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

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

```
                };
        };
```

B.2 Externalization Service

```
// Externalization  Service v1.0 described in CORBAservices:
          //  Common Object Services Specification, chapter 8

            // OMG IDL for CosExternalization Module, p 8-12

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

             interface StreamFactory
             {
                    Stream create();
              };

              interface FileStreamFactory
              {
                     Stream create( in string theFileName)
                            raises( InvalidFileNameError );
              };
            };


            // OMG IDL for CosStream Module, p 8-15

            #include <LifeCycle.idl>
            #include <ObjectIdentity.idl>
```

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

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

          interface StreamableFactory
          {
               Streamable create_uninitialized();
          };


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

B.3 LifeCycle Service

```
// Life Cycle Service v1.0 described in CORBAservices:
          //  Common Object Services Specification, chapter 6

            // OMG IDL for CosLifeCycle Module, p 6-10
            #include "Naming.idl"

            module CosLifeCycle
            {

                 typedef Naming::Name Key;
                 typedef Object Factory;
                 typedef sequence <Factory> Factories;
                 typedef struct NVP
                {
                      Naming::Istring name;
                      any  value;
                 } NameValuePair;
```

```
        typedef sequence <NameValuePair> Criteria;

        exception NoFactory
        {
             Key search_key;
        };
        exception NotCopyable { string reason; };
        exception NotMovable { string reason; };
        exception NotRemovable { string reason; };
        exception InvalidCriteria{
             Criteria invalid_criteria;
        };
        exception CannotMeetCriteria {
             Criteria unmet_criteria;
        };

        interface FactoryFinder
        {
             Factories find_factories(in Key factory_key)
                  raises(NoFactory);
        };

        interface LifeCycleObject
        {
             LifeCycleObject copy(in FactoryFinder there,
                                  in Criteria the_criteria)
                  raises(NoFactory, NotCopyable, InvalidCriteria,
                        CannotMeetCriteria);

             void move(in FactoryFinder there,
                       in Criteria the_criteria)
                  raises(NoFactory, NotMovable, InvalidCriteria,
                        CannotMeetCriteria);
             void remove()
                  raises(NotRemovable);
        };

        interface GenericFactory
           {
             boolean supports(in Key k);
             Object create_object(
                       in Key k,
                       in Criteria the_criteria)
                  raises (NoFactory, InvalidCriteria,
                       CannotMeetCriteria);
           };
   };
```

B.4 Object Transaction Service

```
// Transaction  Service v1.0 described in CORBAservices:
            Common Object Services Specification,
            chapter 10

            // OMG IDL for CosTransactions Module, p 10-65

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

            enum Vote {
                 VoteCommit,
                 VoteRollback,
                 VoteReadOnly
            };

            // Standard exceptions
            exception TransactionRequired {};
            exception TransactionRolledBack {};
            exception InvalidTransaction {};

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

            // Exception from Orb operations
            exception WrongTransaction {};

            // Other transaction-specific exceptions
            exception SubtransactionsUnavailable {};
            exception NotSubtransaction {};
            exception Inactive {};
            exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};

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

// Current transaction pseudo object (PIDL)
interface Current {
     void begin()
          raises(SubtransactionsUnavailable);
     void commit(in boolean report_heuristics)
          raises(
               NoTransaction,
               HeuristicMixed,
               HeuristicHazard
          );
     void rollback()
          raises(NoTransaction);
     void rollback_only()
          raises(NoTransaction);

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

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

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

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

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

interface Coordinator {

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

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

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

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

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

     void rollback_only()
          raises(Inactive);

     string get_transaction_name();

     Control create_subtransaction()
          raises(SubtransactionsUnavailable, Inactive);
};

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

```
};

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

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

interface TransactionalObject {
};

}; // End of CosTransactions Module
```

B.5 Persistent Object Service

```
// Persistent Object Service v1.0 described in
          // CORBAservices: Common Object Services Specification, chapter 5

           // OMG IDL for CosPersistencePID Module, p 5-9

           module CosPersistencePID {

                interface PID {
```

```
          attribute string datastore_type;
          string get_PIDString();
     };

};


// OMG IDL for CosPersistencePO Module, p 5-12

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

module CosPersistencePO {

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

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

// OMG IDL for CosPersistencePOM Module, p 5-15

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

module CosPersistencePOM {

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


     // OMG IDL for CosPersistencePDS Module, p 5-20

     #include "CosPersistencePID.idl"

     module CosPersistencePDS {

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


     // OMG IDL for CosPersistencePDS_DA Module, p 5-22

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

     module CosPersistencePDS_DA {

          typedef string DAObjectID;

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

          interface DAObject {
```

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

     interface DAObjectFactory {
          DAObject create();
     };

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

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


// OMG IDL for CosPersistenceDDO Module, p 5-32

#include "CosPersistencePID.idl"

module CosPersistenceDDO {

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


     // OMG IDL for CosPersistenceDS_CLI Module, p 5-35

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

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

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

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

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

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

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

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

Appendix C

Stream Tunnel Service

Presented here are the IDL definitions for the Stream Tunnel Service and other related services and information. Included are:-

- Stream Tunnel Service IDL
- ExtendedStreamIO IDL
- Tag values for sequence data types used by ExtendedStreamIO

C.1 Stream Tunnel Service

```
#include <CosExternalization.idl>

module StreamTunnelService
{
typedef sequence <any> StreamedData;
typedef CosLifeCycle::Criteria TransferChannelCriteria;

exception ChannelTypeNotSupported{};
exception BaseChannelSupportedOnly{};
exception ChannelOpenFailed{ string reason; };
exception DataTransferError{ string reason; };

exception NoDataAvailable{};
exception StreamNotAvailable{};
exception CannotAcceptData{};

interface StreamChannelServer;
interface StreamChannelFactory;

interface StreamChannel : CosExternalization::Stream
  {
     void pull_stream_data( in StreamChannelServer sourceChannel )
       raises( NoDataAvailable, StreamNotAvailable, DataTransferError);
     void push_stream_data( in StreamChannelServer targetChannel )
       raises( NoDataAvailable, StreamNotAvailable, DataTransferError);
  };


interface StreamChannelServer : StreamChannel
  {
    void push_StreamedData(in StreamedData restoredData)
      raises( CannotAcceptData );
    StreamedData pull_StreamedData()
      raises ( NoDataAvailable, DataTransferError );

    oneway void send_data() raises ( NoDataAvailable, DataTransferError );
    oneway void receive_data() raises( CannotAcceptData );
  };


interface StreamTunnel
  {
    StreamChannel push_streamable( in CosStream::Streamable streamableSource,
                       in StreamChannel targetStream )
      raises( NoDataAvailable, StreamNotAvailable, DataTransferError );

    CosStream::Streamable pull_streamable(in CosLifeCycle::FactoryFinder finder,
                             in StreamChannel sourceStream )
      raises( CosLifeCycle::NoFactory, CosStream::StreamDataFormatError,
            NoDataAvailable, StreamNotAvailable, DataTransferError);


    StreamChannel         open_channel(in         StreamTunnelService::StreamTunnel
other_tunnelEnd,
                          inout TransferChannelCriteria channelType,
                          out StreamChannel otherEnd )
      raises( ChannelTypeNotSupported, BaseChannelSupportedOnly, ChannelOpenFailed
);

    void close_channel( in StreamChannel targetChannel )
      raises( StreamNotAvailable );
  };

interface StreamChannelFactory
```

```
  {
    StreamChannel create(inout TransferChannelCriteria channelType)
      raises( ChannelTypeNotSupported, BaseChannelSupportedOnly, ChannelOpenFailed
);
  };

};
```

C.2 ExtendedStreamIO

```
#include <CosExternalization.idl>


module ExtendedStreamIO
{
typedef sequence <string> StringSeq;
typedef sequence <char> CharSeq;
typedef sequence <octet> OctetSeq;
typedef sequence <unsigned long> UnsignedLongSeq;
typedef sequence <unsigned short> UnsignedShortSeq;
typedef sequence <long> LongSeq;
typedef sequence <short> ShortSeq;
typedef sequence <float> FloatSeq;
typedef sequence <double> DoubleSeq;
typedef sequence <boolean> BooleanSeq;

  interface StreamSeqIO : CosStream::StreamIO
    {
      void write_string_seq(in StringSeq aStringSeq);
      void write_char_seq(in CharSeq aCharSeq);
      void write_octet_seq(in OctetSeq anOctetSeq);
      void write_unsigned_long_seq(in UnsignedLongSeq anUnsignedLongSeq);
      void write_unsigned_short_seq(in UnsignedShortSeq anUnsignedShortSeq);
      void write_long_seq(in LongSeq aLongSeq);
      void write_short_seq(in ShortSeq aShortSeq);
      void write_float_seq(in FloatSeq aFloatSeq);
      void write_double_seq(in DoubleSeq aDoubleSeq);
      void write_boolean_seq(in BooleanSeq aBooleanSeq);

      StringSeq read_string_seq() raises( CosStream::StreamDataFormatError);
      CharSeq read_char_seq() raises( CosStream::StreamDataFormatError );
      OctetSeq read_octet_seq() raises( CosStream::StreamDataFormatError );
      UnsignedLongSeq read_unsigned_long_seq()
             raises( CosStream::StreamDataFormatError );
      UnsignedShortSeq read_unsigned_short_seq()
             raises( CosStream::StreamDataFormatError );
      LongSeq read_long_seq() raises( CosStream::StreamDataFormatError );
      ShortSeq read_short_seq() raises( CosStream::StreamDataFormatError );
      FloatSeq read_float_seq() raises( CosStream::StreamDataFormatError );
      DoubleSeq read_double_seq() raises( CosStream::StreamDataFormatError );
      BooleanSeq read_boolean_seq() raises( CosStream::StreamDataFormatError );
  };
};
```

C.3 Tag Values for ExtendedStreamIO

sequence type	*value*
char sequence	0xe1
octet sequence	0xe2
unsigned long sequence	0xe3
unsigned short sequence	0xe4
long sequence	0xe5
short sequence	0xe6
float sequence	0xe7
double sequence	0xe8
boolean sequence	0xe9
string sequence	0xea

Appendix D

Data Object Service and File Data Object Service

Presented here are the Data Object Service and the extension of it for the access and manipulation of files and directories - File Data Object Service.

D.1 Data Object Service

```
#include <StreamTunnelService.idl>


module DataObjectService
{
typedef string DOb_ID_String;
typedef sequence <DOb_ID_String> DOb_ID_String_Set;

exception DOb_ID_Invalid{ string reason; };
exception DOb_ID_NotFound{};
exception DOb_CreateDenied{ string reason; };
exception DOb_AccessDenied{ string reason; };
exception DOb_UpdateDenied{ string reason; };
exception DOb_RemovalDenied{ string reason; };
exception NoInterfaceMatchingKey{};

interface DataObjectServer;

interface DOb_ID : CosLifeCycle::LifeCycleObject
  {
    attribute DataObjectServer server;
    DOb_ID_String get_stringified_DOb_ID( );
    void set_DOb_ID( in DOb_ID_String DOb_string_identifier )
      raises (DOb_ID_Invalid );
  };


interface DataObjectServer
  {
    StreamTunnelService::StreamTunnel get_StreamTunnel( );
    void create( in DOb_ID_String DOb_identifier )
       raises(DOb_ID_Invalid, DOb_CreateDenied);
    void retrieve( in DOb_ID_String DOb_identifier,
                in StreamTunnelService::StreamChannel transfer_channel )
      raises( DOb_AccessDenied, DOb_ID_NotFound,
            StreamTunnelService::StreamNotAvailable );
    void store( in DOb_ID_String DOb_identifier,
            in StreamTunnelService::StreamChannel transfer_channel )
      raises( DOb_UpdateDenied, DOb_ID_NotFound,
            StreamTunnelService::StreamNotAvailable );
    void remove( in DOb_ID_String DOb_identifier )
      raises( DOb_RemovalDenied, DOb_ID_NotFound );
  };

interface DataObjectManager
{
  attribute DOb_ID target_DataObject;
  void create( ) raises(DOb_ID_Invalid, DOb_CreateDenied);
  void retrieve( )
    raises( DOb_AccessDenied, DOb_ID_NotFound, DOb_ID_Invalid,
          StreamTunnelService::StreamNotAvailable );
  void store( )
    raises( DOb_UpdateDenied, DOb_ID_NotFound, DOb_ID_Invalid,
          StreamTunnelService::StreamNotAvailable );
  void remove( )
    raises( DOb_RemovalDenied, DOb_ID_NotFound, DOb_ID_Invalid );

  void remove_manager( );
};

interface DataObjectManagerFactory
{
  DataObjectManager create( in CosLifeCycle::Key manager_interface_type,
                      in DOb_ID initial_DOb_identifier)
    raises (DOb_ID_Invalid,NoInterfaceMatchingKey);
};

};
```

D.2 File Data Object Service

```
#include <DataObjectService.idl>
#include <CosExternalization.idl>

module FileDObService
{
typedef sequence<octet> Bytes;

enum ftype{ FILE, DIRECTORY };
struct Content
{
  string fname;
  ftype file_type;
};
typedef sequence <Content> Contents;

exception IOException{};
exception EndOfFile{};
exception NoDirectoryDataAvailable{};


//DirContents for private use of DirectoryDObManager
interface DirContents : CosStream::Streamable, CosLifeCycle::LifeCycleObject
  {
    attribute Contents fileList;
  };

//BLOb interface for private use of FileDObManager
interface BLOb : CosStream::Streamable, CosLifeCycle::LifeCycleObject
```

```
  {
    attribute Bytes BLOb_data;
    attribute long sliceStart;
    attribute long sliceSize;

    void add_slice(in Bytes part_of_file) raises(IOException);
    Bytes take_slice() raises(EndOfFile);
  };


interface DirectoryDOb_ID : DataObjectService::DOb_ID
  {
    attribute string fpath;
  };

interface FileDOb_ID : DirectoryDOb_ID
  {
    attribute string fname;
    attribute long sliceStart;
    attribute long sliceSize;
  };

interface DirectoryDObManager : DataObjectService::DataObjectManager
  {

    Contents get_contents() raises(NoDirectoryDataAvailable);
  };

interface FileDObManager : DataObjectService::DataObjectManager
  {
    long getFilePointer() raises(IOException);
    void seek(in long offset) raises(IOException);
    long length() raises(IOException);
    void setlength(in long fileLength) raises(IOException);

    octet readByte() raises(EndOfFile,IOException);
    Bytes readBytes(in long NoOfBytes) raises(EndOfFile,IOException);
    char readChar() raises(EndOfFile,IOException);
    string readString() raises(EndOfFile,IOException);
    boolean readBoolean() raises(EndOfFile,IOException);
    short readShort() raises(EndOfFile,IOException);
    unsigned short readUShort() raises(EndOfFile,IOException);
    long readLong() raises(EndOfFile,IOException);
    unsigned long readULong() raises(EndOfFile,IOException);
    float readFloat() raises(EndOfFile,IOException);
    double readDouble() raises(EndOfFile,IOException);

    void writeByte(in octet aByte) raises(IOException);
    void writeBytes(in Bytes someBytes) raises(IOException);
    void writeChar(in char aChar) raises(IOException);
    void writeString(in string aString) raises(IOException);
    void writeBoolean(in boolean aBoolean) raises(IOException);
    void writeShort(in short aShort) raises(IOException);
    void writeUShort(in unsigned short aUShort) raises(IOException);
    void writeLong(in long aLong) raises(IOException);
    void writeULong(in unsigned long aULong) raises(IOException);
    void writeFloat(in float aFloat) raises(IOException);
    void writeDouble(in double aDouble) raises(IOException);
  };

interface StreamIOFileDObManager
      : DataObjectService::DataObjectManager
  {
    CosStream::StreamIO getStreamIOInterface();
  };
};
```

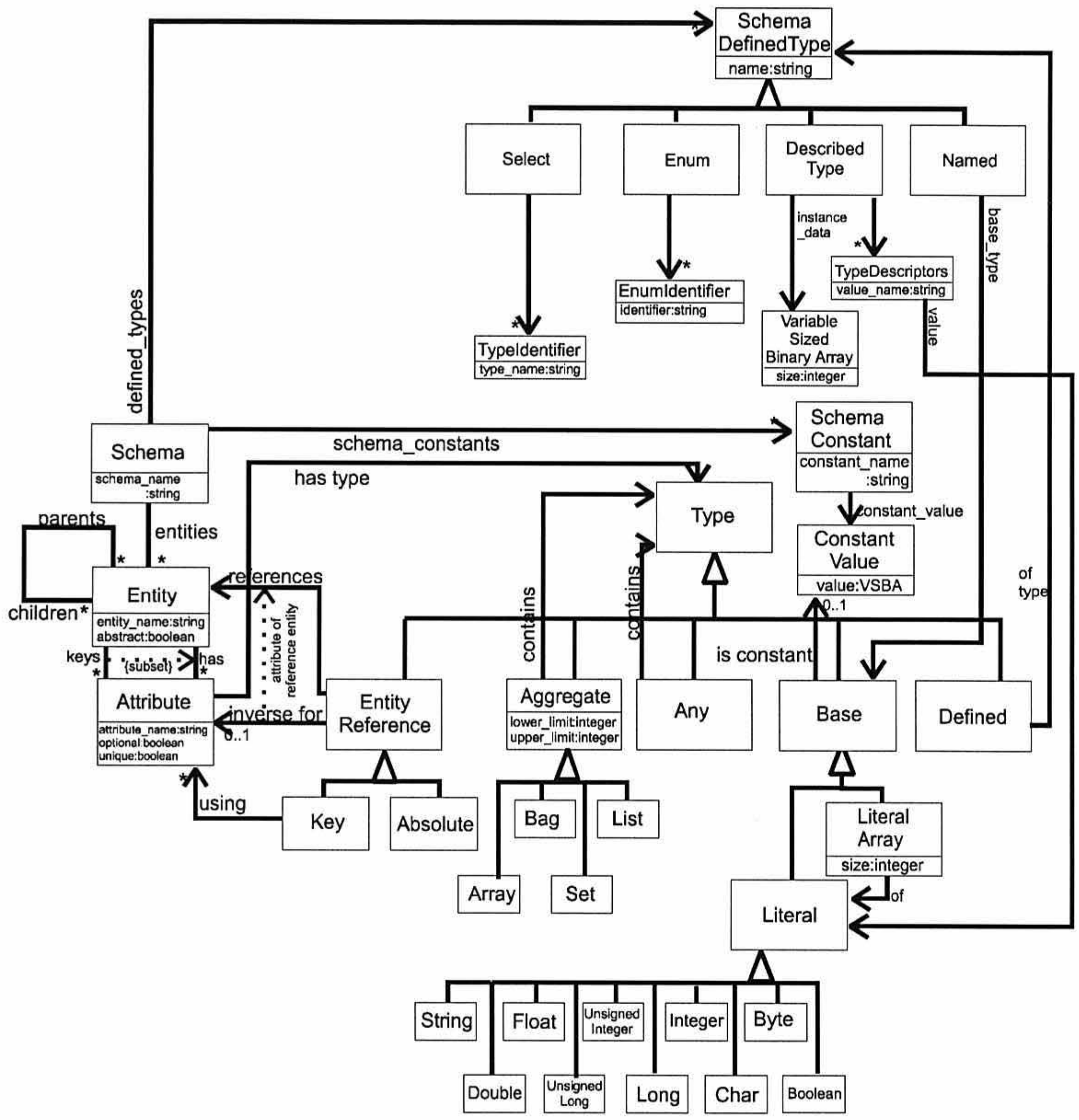
Appendix E

Meta-Schema Model

Presented here are material relevant to the Meta-Schema Model, including:-

- Meta-Schema Model diagram
- Meta-Schema Model IDL
- Meta-Schema Facility IDL
- Meta-Schema Stream Format syntax

E.1 Meta-Schema Model



E.2 Meta-Schema Model IDL

```
module MSModel
{
   typedef string Identifier;
   typedef sequence <Identifier> Identifiers;

   exception NotFound{};

   interface msEntity;
   typedef sequence <msEntity> msEntities;
   interface msSchemaDefinedType;
   typedef sequence <msSchemaDefinedType> msSchemaDefinedTypes;
   interface msSchemaConstant;
   typedef sequence <msSchemaConstant> msSchemaConstants;
   interface msAttribute;
   typedef sequence <msAttribute> msAttributes;
   interface msType;
   interface msLiteral;
   interface msSchemaDefinedType;
   interface msConstantValue;


   interface msSchema
      {
       readonly attribute Identifier schema_name;
       readonly attribute msEntities entities;
       readonly attribute msSchemaDefinedTypes defined_types;
       readonly attribute msSchemaConstants constants;

       msEntity find_entity(in Identifier entity_name) raises(NotFound);

       msSchemaDefinedType find_schema_defined_type(
          in Identifier schema_defined_type_name) raises(NotFound);

       msSchemaConstant find_schema_constant(
          in Identifier schema_constant_name) raises(NotFound);
      };

    interface msEntity
       {
        readonly attribute Identifier entity_name;
        readonly attribute boolean is_abstract;
        readonly attribute Identifiers parents;
        readonly attribute Identifiers children;
        readonly attribute msAttributes attributes;
        readonly attribute msAttributes keys;

        msAttribute find_attribute(in Identifier attribute_name)
             raises(NotFound);
       };

    interface msAttribute
       {
        readonly attribute Identifier attribute_name;
        readonly attribute boolean is_optional;
        readonly attribute boolean is_unique;
        readonly attribute msType type_of;
       };

     interface msType
       {
       };

      interface msEntityReference : msType
      {
         readonly attribute boolean is_inverse;
         readonly attribute Identifier referenced_entity;
      readonly attribute Identifier inverse_attribute;
   };

   interface msKey : msEntityReference
   {
      readonly attribute Identifiers key_attributes;
   };

   interface msAbsolute : msEntityReference
   {
   };

   interface msAggregate : msType
   {
      readonly attribute long lower_limit;
      readonly attribute long upper_limit;
      readonly attribute msType contains_type;
   };

   interface msArray : msAggregate
   {
   };

   interface msBag : msAggregate
   {
   };

   interface msSet : msAggregate
   {
   };

   interface msList : msAggregate
   {
   };

   interface msAny : msType
   {
   };

   interface msBase : msType
   {
      readonly attribute boolean is_constant;
      readonly attribute msConstantValue constant_value;
   };

   interface msLiteralArray : msBase
   {
      readonly attribute long array_size;
      readonly attribute msLiteral array_of;
   };

   interface msLiteral : msBase
   {
      enum LiteralType { String, Double, Float, UnsignedLong, UnsignedInteger,
                         Long, Integer, Character, Byte, Boolean };

      readonly attribute LiteralType literal_type;
   };

   interface msDefined : msType
   {
      readonly attribute Identifier type_of;
   };

   interface msSchemaDefinedType
   {
      readonly attribute Identifier schema_defined_type_name;
   };

   interface msNamed : msSchemaDefinedType
```

```
        {
           readonly attribute msBase base_type;
        };

       interface msSelect : msSchemaDefinedType
        {
           readonly attribute Identifiers type_identifiers;
        };

       interface msEnum : msSchemaDefinedType
        {
           readonly attribute Identifiers EnumIdentifiers;
        };

       interface msDescribedType : msSchemaDefinedType
        {
           struct TypeDescriptor
           {
               string value_name;
               any value;
           };

           typedef sequence <TypeDescriptor> TypeDescriptors;

           readonly attribute TypeDescriptors type_descriptors;
        };

       interface msConstantValue
        {
           readonly attribute any value;
        };

       interface msSchemaConstant
        {
           readonly attribute Identifier constant_name;
        };

};
```

E.3 Meta-Schema Facility

```
#include "MSModel.idl"
#include "CosExternalization.idl"
#include "CosLifeCycle.idl"

module MSF
{
   typedef string Identifier;
   typedef sequence <Identifier> Identifiers;

   exception NotFound{};
   exception MetaInfoError{ string explanation; };

   interface MetaSchemaBuilderFactory;

   interface MetaSchemaRespository : CosLifeCycle::LifeCycleObject
      {
       Identifiers get_schema_list();
       Identifiers get_entity_list(in Identifier schema_name) raises(NotFound);
       Identifiers get_schema_defined_type_list(in Identifier schema_name)
                                                    raises(NotFound);
       Identifiers get_schema_constant_list(in Identifier schema_name)
                                                    raises(NotFound);

       CosStream::Stream
```

```
            get_schema_meta_info(in Identifier schema_name) raises(NotFound);

        CosStream::Stream
            get_entity_meta_info(in Identifier schema_name,
                            in Identifier entity_name) raises(NotFound);

        CosStream::Stream
            get_schema_defined_type_meta_info(in Identifier schema_name,
                                    in Identifier defined_type_name)
                                    raises(NotFound);

        CosStream::Stream
            get_schema_constant_meta_info(in Identifier schema_name,
                                 in Identifier constant_name) raises(NotFound);

       };


    interface MetaSchemaBuilderFactory
      {
        MSModel::msSchema create_schema_meta_model(
                          in CosStream::StreamIO schema_meta_info)
                          raises(MetaInfoError);

        MSModel::msEntity create_entity_meta_model(
                          in CosStream::StreamIO schema_meta_info)
                          raises(MetaInfoError);

        MSModel::msSchemaDefinedType create_schema_defined_type_meta_model(
                          in CosStream::StreamIO schema_meta_info)
                          raises(MetaInfoError);

        MSModel::msSchemaConstant create_schema_constant_meta_model(
                          in CosStream::StreamIO schema_meta_info)
                          raises(MetaInfoError);
       };

};
```

E.4 Meta-Schema Stream Format Syntax

Tag Value (*short*)	**Tag Name**	**Syntax**
1	Schema	schema_name:string [no_of_schema_constants:integer schema_constants] [no_of_defined_types:integer defined_types] no_of_entities entities
2	schema_constants	Schema_constant [schema_constants]
3	SchemaConstant	name:string type:Base Constant_value
4	ConstantValue	value:VSBA
5	defined_types	SchemaDefinedType [defined_types]
6	SchemaDefinedType	type_name:string (Select | Enum | Named | DescribedType)
7	Select	no_of_type_identifiers:integer type_identifiers
8	type_identifiers	TypeIdentifier [type_identifiers]
9	TypeIdentifier	string
10	Enum	no_of_enumIdentifiers:integer enumIdentifiers
11	enum_identifiers	Enum_identifier [enum_identifiers]
12	EnumIdentifierstring	

13	Named	Base
14	DescribedType	no_of_type_descriptors:integer type_descriptors VSBA
15	type_descriptors	TypeDescriptor [type_descriptors]
16	TypeDescriptor	value_name value:Base
17	entities	Entity [entities]
18	Entity	entity_name abstract:boolean [no_of_parents:integer parents] [no_of_children:intger children] no_of_attributes attributes [no_of_attribute_keys attribute_keys]
19	parents	parent [parents]
20	parent	entity_name
21	children	child [children]
22	child	entity_name
23	entity_name	string
24	attributes	Attribute [attributes]
25	Attribute	attribute_name optional:boolean unique:boolean Type
26	attribute_name	string
27	attribute_keys	attribute_key [attribute_keys]
28	attribute_key	attribute_name
29	Type	EntityReference | Aggregate | Any | Defined | Base | UndefinedType
30	types	Type [types]
31	EntityReference	references_entity [inverse_attribute] (Key | Absolute)
32	references_entity	entity_name
33	inverse_attribute	attribute_name
34	Key	attribute_keys
35	Absolute	
36	Aggregate	lower_limit:integer upper_limit:integer aggregate_type Type
37	aggregate_type	Array | Bag | Set | List
38	Array	
39	Bag	
40	Set	
41	List	
42	Any	
43	Base	[Constant_value] (Literal | LiteralArray)
44	Literal	String | Double | Float | Unsigned_long | Unsigned_integer | Long | Integer| Character | Byte | Boolean
45	LiteralArray	size:integer Literal
46	VSBA[1]	size:integer
47	Defined	SchemaDefinedType
48	String	
49	Double	
50	Float	
51	UnsignedLong	

[1] VSBA-Variable Sized Binary Array

52	UnsignedInteger
53	Long
54	Integer
55	Character
56	Byte
57	Boolean

Appendix F

Persistent Data Access Service

Presented here is material related to the Persistent Data Access Service, including:-

- Persistent Data Access Service
- Entity Data Object Service
- Variable Sized Binary Array Manager Interface
- Retrieval Map
- Structured Entity Stream Format syntax

F.1 Persistent Data Access Service

```
module PDAS
{
// PDAS - Persistent Data Object Service
exception InvalidAttribute{};
exception IllegalCast{};
exception InvalidEnumId{};
exception InvalidSelectType{};
exception NullReference{};
exception InvalidRefenceAssignment{};
exception NotFound{};
exception CreateFailure{CosPropertyService::Properties reason, any further_info};
exception DeleteFailure{CosPropertyService::Properties reason, any further_info};
exception StoreFailure{CosPropertyService::Properties reason, any further_info};
exception ExactCopiesNotSupportedByModel{};
exception TransactionInPrograss();

interface EntityReference;
typedef sequence <EntityReference> EntityReferences;

interface PDASServer::DatastoreServer;
interface VSBA::VSBAManager;
```

```
// The client session interface allowing entity instance creation/deletion/copying,
// access to Meta-Schema information on data, querying and static binding of CORBA
// objects.

interface Session   : CosConcurrencyControl::LockSet
       {
             EntityReference create(in string entity_type_name)
                            raises(NotFound,CreateFailure);
             EntityReference copy(in EntityReference source_entity)
                          raises(CreateFailure,ExactCopiesNotSupportedByModel);
             void delete(in EntityReference entity) raises(DeleteFailure);

             void save_all_updates( ) raises(StoreFailure);

             MSModel::msSchema get_schema_model( );
             RetrievalModel::RetrievalMaps get_retrieval_maps( );

             CosQuery::QueryEvaluator get_query_evaluator( );

             Object bind_object(in EntityReference er,
                                in CosLifeCycle::FactoryFinder finder)
                                raises(CosLifeCycle::NoFactory);

             EntityReference find_entity_by_oid(in long oid) raises(NotFound);

             void close_session( ) raises(TransactionInProgress);
   };

interface NonTransactionalSession : Session
       {
       };

interface TransactionalSession : Session,
                            CosTransactions::TransactionalObject
   {
   };

// SessionFactory interface for creating and initialising a client session

interface SessionFactory
       {
             Session create(in PDASServer::DatastoreServer datastore_server,
                          in string datastore_name,
                          in boolean transactional_session,
                          in CosConcurrencyControl::lock_mode initial_lock_mode,
                          in CosPropertyService::Properties
                          initialisation_attributes)
                          raises(CreateFailure);
       };

// EntityReference interface represent a handle to a entity instance,
// allowing manipulation of its attributes and gaining information of
// on the type of entity.

interface EntityReference
       {
             boolean is_same(EntityReference other_entity);
             boolean is_kind_of(string entity_type_name);

             MSModel::msEntity get_entity_type();
             MSModel::msType get_attribute_type(in string attr_name)
                    raises(InvalidAttribute);

             long get_entity_session_oid( );

             void save_updates( ) raises(StoreFailure);

             void release( );

             // Operations to set the values of entity attributes
```

```
void set_String(in string attr_name, in string s)
       raises(InvalidAttribute,IllegalCast);
void set_Double(in string attr_name, in double d)
       raises(InvalidAttribute,IllegalCast);
void set_Float(in string attr_name, in float f)
       raises(InvalidAttribute,IllegalCast);
void set_UnsignedLong( in string attr_name, in unsigned long ul)
       raises(InvalidAttribute,IllegalCast);
void set_UnsignedInteger( in string attr_name, in unsigned short ui)
       raises(InvalidAttribute,IllegalCast);
void set_Long(in string attr_name, in long l)
       raises(InvalidAttribute,IllegalCast);
void set_Integer(in string attr_name, in short i)
       raises(InvalidAttribute,IllegalCast);
void set_Char(in string attr_name, in char c)
       raises(InvalidAttribute,IllegalCast);
void set_Byte(in string attr_name, in octet o)
       raises(InvalidAttribute,IllegalCast);
void set_Boolean(in string attr_name, in boolean b)
       raises(InvalidAttribute,IllegalCast);
void set_LiteralArray(in string attr_name, in any la)
       raises(InvalidAttribute,IllegalCast);
void set_Enum(in string attr_name, in short enum_id)
       raises(InvalidAttribute,IllegalCast,InvalidEnumId);
void set_Select(in string attr_name, in any s)
       raises(InvalidAttribute,IllegalCast,InvalidSelectType);
void set_Any(in string attr_name, in any a)
       raises(InvalidAttribute,IllegalCast);
void set_Aggregate(in string attr_name, in CosCollection::Collection
c)
                                   raises(InvalidAttribute,IllegalCast);
void set_EntityReference(in string attr_name, in EntityReference er)
       raises(InvalidAttribute,IllegalCast,InvalidRefenceAssignment);

// Operations to get the values of entity attributes
string get_String(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
double get_Double(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
float get_Float(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
unsigned long get_UnsignedLong(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
unsigned short get_UnsignedInteger(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
long get_Long(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
short get_Integer(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
char get_Char(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
octet get_Byte(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
boolean get_Boolean(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
any get_LiteralArray(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
short get_Enum(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
any get_Select(in string attr_name)
       raises(InvalidAttribute,IllegalCast);
any get_Any(in string attr_name)
       raises(InvalidAttribute,IllegalCast);

CosCollection::Collection get_Aggregate(in string attr_name)
       raises(InvalidAttribute,IllegalCast);

VSBA::VSBAManager get_DescribedType_data(in string attr_name)
      raises(InvalidAttribute,IllegalCast);
```

```
            EntityReference get_Entity(in string attr_name)
                    raises(InvalidAttribute,IllegalCast,NullReference);

            EntityReference get_Entity_using_map(in string attr_name,
                                                RetrivalMap map_name)
                    raises(InvalidAttribute,IllegalCast,NullReference);
   };

};

// PDASServer module contains all interfaces concerned with the
// server side of the Persistent Data Access service.

module PDASServer
{
      typedef sequence <octet> BinaryData;
      typedef sequence <long> OIDs;

      interface ServerSession;

      //interface to create and start a server session in the server process
      interface DatastoreServer
            {
       ServerSession start_server_session(in string datastore_name,
                  in boolean transactional_session,
                  in CosConcurrencyControl::lock_mode initial_lock_mode,
                  in CosPropertyService::Properties initialisation_attributes)
                     raises(PDAS::CreateFailure);
            };

      //The server side session
      interface ServerSession : CosConcurrencyControl::LockSet
            {
                  DataObjectService::DataObjectServer get_data_object_server( );

                  MSModel::msSchema get_meta_schema_model( );
                  MSR::MetaSchemaRespository get_meta_schema_repository( );

                  RetrievalModel::RetrievalMaps get_retrieval_maps( );

                  CosQuery::QueryEvaluator get_server_query_evaluator( );

                  void close_session( ) raises(PDAS::TransactionInProgress);
            };

      interface NonTransactionalServerSession : ServerSession
            {
            };

      interface TransactionalServerSession : ServerSession
                                    CosTransactions::Resource,
                                     CosConcurrency::LockCoordinator
            {
            };
};
```

F.2 Entity Data Object Service

```
// The EntityDObService module contains the interfaces that extend
// the Data Object Service to allow access and management of entity instances.

module EntityDObService
{
        typedef sequence <long> oid_array;
        // DataObjectService extension interfaces

        // Data Object Identifier Extensions
   interface Entity_DOb_ID : DataObjectService::DOb_ID
      {
                attribute long oid;
                attribute string attr_name;
      };

   interface OID_Array : DataObjectService::DOb_ID
        {
                oid_array OIDs;
        };

        interface EntityGraph_DOb_ID : Entity_DOb_ID
        {
                attribute string retrival_map;
        };

   interface Aggregate_DOb_ID : Entity_DOb_ID
        {
        };

   interface AggregateEntity_DOb_ID : Aggregate_DOb_ID
        {
                attribute long entity_number;
        };

   interface DescribedType_DOb_ID : Entity_DOb_ID
        {
        };

// Data Object Manager Extension
   interface EntityDObManager : DataObjectService::DataObjectManager
        {
                void store_Entity(in PDAS::EntityReference er);

                void store_Entities(in PDAS::EntityReferences ers);

                void store_EntityGraph(in PDAS::EntityReference root_er);

                void store_Aggregate(in CosCollection::Collection c);

                void store_DescribedType(in BinaryData data);

                PDAS::EntityReference retrieve_Entity( ) raises(PDAS::NotFound);

                PDAS::EntityReferences retrieve_Entities( ) raises(PDAS::NotFound);

                PDAS::EntityReferences retrieve_EntityGraph( ) raises(PDAS::NotFound);

                CosCollection::Collection retrieve_Aggregate( )
raises(PDAS::NotFound);

                BinaryData retrieve_DescribedType( ) raises(PDAS::NotFound);
        };

   // Streamable interface used as a container for multiple:- Entity instances,
   // Explicit Aggregates and Explicit DescribedTypes.

interface EntityContainer : CosStream::Streamable, CosLifeCycle::LifeCycleObject
```

```
         {
         void add_Entity(in EntityReference er);
         void add_Aggregate(in long entity_oid,
                            in string attr_name,
                            in CosCollection::Collection c);
         void add_DescribedType(in long entity_oid,
                            in string attr_name,
                            in BinaryData data);

         PDAS::EntityReference get_Entity(in long oid)raises(PDAS::NoFound);
               CosCollection::Collection get_Aggregate(in long entity oid,
                                                    in string attr_name)
                                   raises(PDAS::NotFound);

         BinaryData get_DescribedType(in long entity oid,
                                   in string attr_name)
                                raises(PDAS::NotFound);
         };

     interface EntityContainerFactory
         {
               EntityContainer create(MSModel::msSchema schema_model);
         };

// StructuredEntityStreamIO used for wrapping the low level StreamIO interface.
// This interface adds to the Stream, tag information to maintain the structure
// of entities in their serialised form.

     exception StructuredStreamFormatError;

     interface StructuredEntityStreamIO
         {
               void write_EntityIdentifier(in long oid);

               void write_StringAttribute(in string value);
               void write_DoubleAttribute(in double value);
               void write_FloatAttribute(in float value);
               void write_UnsignedLongAttribute(in unsigned long value);
               void write_UnsignedIntegerAttribute(in unsigned short value);
               void write_LongAttribute(in long value);
               void write_IntegerAttribute(in short value);
               void write_CharAttribute(in char value);
               void write_ByteAttribute(in octet value);
               void write_BooleanAttribute(in boolean value);
               void write_LiteralArrayAttribute(
                            in MSModel::msLiteral::LiteralType element_type,
                            in any value);
               void write_EnumAttribute(in short value);
               void write_SelectAttribute(in any value);
               void write_AnyAttribute(in any value);

               void write_AggregateAttribute(in long number_of_elements,
                                          in boolean write_as_block);
               void write_AggregateElement(in long element_number,
                                          in any element);
               void write_AggregateBlock(in any elements);

               void write_DescribedType(in BinaryData data);

               void write_EntityReferenceAttribute(in long oid);
               void write_NullEntityReferenceAttribute( );

               void write_ExplicitAggregate(in long oid, in string attr_name);

               void write_ExplicitDescribedType(in long oid,
                                   in string attr_name,
                                   in BinaryData data);
```

```
                  short read_contained_tag( )
raises(StructuredStreamFormatError);

            long read_EntityIdentifier( ) raises(StructuredStreamFormatError);

            string read_StringAttribute( ) raises(StructuredStreamFormatError);
            double read_DoubleAttribute( ) raises(StructuredStreamFormatError);
            float read_FloatAttribute( ) raises(StructuredStreamFormatError);
            unsigned long read_UnsignedLongAttribute( )
                                          raises(StructuredStreamFormatError);
            unsigned short read_UnsignedIntegerAttribute( )
                                          raises(StructuredStreamFormatError);
            long read_LongAttribute( ) raises(StructuredStreamFormatError);
            short read_IntegerAttribute( ) raises(StructuredStreamFormatError);
            char read_CharAttribute( ) raises(StructuredStreamFormatError);
            octet read_ByteAttribute( ) raises(StructuredStreamFormatError);
            boolean read_BooleanAttribute( ) raises(StructuredStreamFormatError);
            any read_LiteralArrayAttribute(
                          out MSModel::msLiteral::LiteralType element_type )
                                raises(StructuredStreamFormatError);
            short read_EnumAttribute( ) raises(StructuredStreamFormatError);
            any read_SelectAttribute( ) raises(StructuredStreamFormatError);
            any read_AnyAttribute( ) raises(StructuredStreamFormatError);
            long read_AggregateAttribute(out boolean read_as_block,
                               out boolean ExplicitRetrieval)
                               raises(StructuredStreamFormatError);
            any read_AggregateElement(out long element_number)
                             raises(StructuredStreamFormatError);
            any read_AggregateBlock( ) raises(StructuredStreamFormatError);
            BinaryData read_DescribedType(out boolean ExplicitRetrieval)
                             raises(StructuredStreamFormatError);
            long read_EntityReferenceAttribute( )
                               raises(StructuredStreamFormatError);
            void read_ExplicitAggregate(out long oid, out string attr_name)
                                raises(StructuredStreamFormatError);
            BinaryData read_ExplicitDescribedType(out long oid,
                                          out string attr_name)
                                          raises(StructuredStreamFormatError);

            void remove( );
            };

       interface StructuredEntityStreamIOFactory
            {
                  StructuredEntityStreamIO create(in CosStream::StreamIO
basicIO);
            };

};
```

F.3 Variable Sized Binary Array Manager

The Variable Sized Binary Array(VSBA) Manager is an interface for the manipulation of data contained in a described type value, where the structure of the data is explicitly known by the application.

```
module VSBA
{
// VSBA - Variable Sized Binary Array
exception EndOfArray{};

// Interface to access the binary data of DescribedTypes
interface VSBAManager
   {
       long getArrayPointer();
       void seek(in long offset) raises(EndOfArray);
       long length();
       void setlength(in long ArrayLength);

       octet readByte() raises(EndOfArray);
       Bytes readBytes(in long NoOfBytes) raises(EndOfArray);
       char readChar() raises(EndOfArray);
       string readString() raises(EndOfArray);
       boolean readBoolean() raises(EndOfArray);
       short readShort() raises(EndOfArray);
       unsigned short readUShort() raises(EndOfArray);
       long readLong() raises(EndOfArray);
       unsigned long readULong() raises(EndOfArray);
       float readFloat() raises(EndOfArray);
       double readDouble() raises(EndOfArray);

       void writeByte(in octet aByte);
       void writeBytes(in Bytes someBytes);
       void writeChar(in char aChar);
       void writeString(in string aString);
       void writeBoolean(in boolean aBoolean);
       void writeShort(in short aShort);
       void writeUShort(in unsigned short aUShort);
       void writeLong(in long aLong);
       void writeULong(in unsigned long aULong);
       void writeFloat(in float aFloat);
       void writeDouble(in double aDouble);
   };
};
```

F.4 Retrieval Map

```
module RetrievalModel
{
       interface RetrievalMap;
       typedef sequence <RetrievalMap> Maps

       interface Node;

       interface TraverseRelationship;
       typedef sequence <TraverseRelationship> TraverseRelationships;

       interface RetrievalMaps
              {
                     attribute Maps maps;
                     RetrievalMap get_map(in string map_name);
              };

       interface RetrievalMap
              {
                     attribute string map_name;
                     attribute Node root_entity;
              };

       interface Node
              {
                     attribute MSModel::msEntity associated_entity_type;
                     attribute TraverseRelationships attributes;
              };

       interface TravserseRelationship
              {
                     attribute MSModel::msAttribute referenced_attribute;
                     attribute boolean referenced_attribute_has_a_node;
```

```
                attribute Node node;
        };
};
```

F.5 Structured Entity Stream Format Syntax

Tag Value (*short*)	**Tag Name**	**Syntax**
1.	Contents	ContainedItem [Contents]
2.	ContainedItem	EntityIdentifier | ExplicitAggregate | ExplicitDescribedType
3.	EntityIdentifier	*entity_type_name(string) oid_value(long)* Attributes
4.	Attributes	Attribute [Attributes]
5.	Attribute	StringAttribute | DoubleAttribute | FloatAttribute | UnsignedLongAttribute | UnsignedIntegerAttribute | LongAttribute | IntegerAttribute | CharAttribute | ByteAttribute | BooleanAttribute | LiteralArrayAttribute | EnumAttribute | SelectAttribute | AnyAttribute | AggregateAttribute | DescribedTypeAttribute | EntityReferenceAttribute
6.	StringAttribute	*value(string)*
7.	DoubleAttribute	*value(double)*
8.	FloatAttribute	*value(float)*
9.	UnsignedLongAttribute	*value(unsigned long)*
10.	UnsignedIntegerAttribute	*value(unsigned short)*
11.	LongAttribute	*value(long)*
12.	IntegerAttribute	*value(short)*
13.	CharAttribute	*value(char)*
14.	ByteAttribute	*value(octet)*
15.	BooleanAttribute	*value(boolean)*
16.	LiteralArrayAttribute	LiteralType *number_of_elements(long) values*
17.	EnumAttribute	*value(short)*
18.	SelectAttribute	AnyValue
19.	AnyAttribute	AnyValue
20.	AggregateAttribute	explcit_retrieval(*boolean*) [*number_of_elements(long) write_as_block(boolean)* AggregateElement | AggregateBlock]
21.	AggregateElement	AnyValue [AggregateElement]
22.	AggregateBlock	AnyType *values*
23.	DescribedTypeAttribute	explcit_retrieval(*boolean*) [*value(seq. of octet)*]
24.	EntityReference	*oid(short)* | *-1(short)* "null reference"
25.	ExplicitAggregate	*oid(short) attribute_name(string)* AggregateAttribute
26.	ExplicitDescribedType	*oid(short) attribute_name(string)* DescribedTypeAttribute
27.	LiteralType	StringType | DoubleType | FloatType | UnsignedLongType | UnsignedIntegerType | LongType | IntegerType | CharType | ByteType | BooleanType
28.	AnyValue	AnyType *value*
29.	AnyType	StringType | DoubleType | FloatType | UnsignedLongType | UnsignedIntegerType | LongType | IntegerType | CharType | ByteType | BooleanType | LiteralArrayType | EnumType | AggregateType | EntityReferenceType
30.	StringType	
31.	DoubleType	
32.	FloatType	
33.	UnsignedLongType	
34.	UnsignedIntegerType	
35.	LongType	
36.	IntegerType	
37.	CharType	
38.	ByteType	
39.	BooleanType	
40.	LiteralArrayType	
41.	EnumType	
42.	AggregateType	
43.	EntityReferenceType	

References

[Amar] V.Amar, A.Zarli. *Linking STEP and CORBA standards for applications interoperability.* Centre Scientifique et Technique du Batiment, France. http://cic.cstb.fr/ilc/.

[Ambler 99] S.W.Ambler. *Mapping Objects To Relational Databases(White paper).* AmbySoft Inc., http://www.AmbySoft.com/.

[Amirb 97] V.Amirbekyan, K.Zielinski. *What CORBA/ODB integration technique to choose: Adaptor vs. Wrapper.* Workshop #21: Experiences Using Object Data Management in the Real-World, OOPSLA'97, Atlanta, Georgia, USA, Oct. 1997.

[Baker 97] S.Baker. *CORBA Distributed Objects Using Orbix.* ACM Press, Addison Wesley, 1997.

[Ball 98] C.H.Ball, S.Hope. *Data Access and Transportation Services for the CORBA Environment.* TOOLS '98 Conference, Santa Barbara, USA, Aug. 1998

[Ball 99] C.H.Ball, S.Hope. *A Framework of Data Access Services as a necessary alternative to Object Persistence to provide access to Persistent Data for CORBA.* Uni. Of Wales, Bangor, U.K., 1999. ftp://ftp.sees.bangor.ac.uk/craig/.

[Bernstein 97] P.A.Bernstein, E.Newcomer. *Principles of Transaction Processing for the Systems Professional.* Morgan Kaufmann, 1997.

[Chorus 97] Chorus Systems. COOL-ORB v4.1 Programmer's Manual.

[Coad 90] P.Coad, E.Yourdon. *Object-Oriented Analysis.* Yourdon Press, 1990.

[Coad 91] P.Coad, E.Yourdon. *Object-Oriented Design.* Yourdon Press, 1991.

[Coulouris 88] G.F.Couloris, J.Dollimore. *Distributed Systems Concepts and Design.* Addison Wesley, 1988.

[DBTools 98] Rogue Wave Software. *DBTools.h++.* Rogue Wave Software, Inc., Boulder, Colarado, USA, 1998.

[EXPRESS 92] ISO 10303, *Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 11. Description Methods: The EXPRESS Language Reference Manual,* ISO TC 184/SC4, 1992.

[Fleming 97] K.Fleming, S.Aslam-Mir, J.Damstra, M.Vilicich. *Distributed Transactions using CORBA.* Expersoft Corporation.

[Fowler 96] J.Fowler. *STEP for Data Management, Exchange and Sharing.* Technology Appraisals, 1996.

[Godfrey 97] M.Godfrey. *The OpenSpirit E&P Component Framework - A White Paper.* PrismTech, Houston, Texas, USA. http://www.openspirit.com/

[Grasso 96] E.Grasso. *Passing Objects by Value in CORBA.* OMG Document orbos/96-07-03.

[Grasso 97] E.Grasso, N.Perdigues-Charton. *Transaction Concepts in Connection Management Applications.* TINA Conference 97, Santiago, Chile, Nov. 17-21, 1997.

[Grasso 97b] E.Grassio. *Implementing Interposition in CORBA Object Transaction Service.* 1st International Enterprise Distributed Object Computing, Gold Coast, Australia, 24-26 October 1997.

[Joseph 90] J.V.Joseph, S.M.Thatte, C.W.Thompson, D.L. Wells. *Object-Oriented Databases:Design and Implementation.* Proceedings of the

IEEE. Vol. 79, no. 1, p 42-62, Jan 91.

[JDK] Sun Microsystems. *Java Development Kit v1.1.* Sun Microsystems Inc., http://www.javasoft.com/products/jdk/1.1/.

[JIDL] Sun Microsystems. *JavaIDL Object Request Broker.* Sun Microsystems Inc., http://www.javasoft.com/products/jdk/idl/index.html.

[Kim 95] W.Kim (ed.). *Modern Database Systems - The Object Model, Interoperability, and Beyond.* Addison Wesley, 1995.

[Klein 95] J.Kleindienst, F.Plasil, P.Tuma. *Implementing CORBA Persistence Service.* Tech. Report No. 117, Charles Uni., Prague, Czech Republic, Dec. 1995.

[Klein 96] J.Kleindienst, F.Plasil, P.Tuma. *Lessons Learnt from Implementing the CORBA Persistence Service.* OOPSLA '96, San Jose, Oct. 1996.

[Klein 96b] J.Kleindienst, F.Plasil, P.Tuma. *What We Are Missing in the CORBA Persistent Object Service Specification.* Charles Uni., Prague, Czech Republic, 1996.

[McFadden 94] F.R.McFadden, J.A.Hoffer. *Modern Database Management.* Addison Wesley, 1994.

[ODMG 93] Object Database Management Group. *The Object Database Standard: ODMG-93 Rel. 1.1.* R.G.G.Cattell(ed.), Morgan Kaufmann, 1994.

[OOSA 97] IONA Technologies. *Orbix+ObjectStore Adaptor(White Paper).* IONA Technologies Ltd., Dublin, Ireland, April 1997.

[OMAG 90] Object Management Group. *The Object Management Architecture Guide*: Object Management Group, Inc., Framingham, MA. Nov 90.

[OMG 95] Object Management Group. *The Common Object Request Broker Architecture and Specification; Revision 2.0.* OMG 96-3-4. Object

Management Group, Inc., Framingham, MA., July 1995.

[OMG 95b] Object Management Group. *Object Services RFP 5.* OMG TC Document 95-3-25, 1995.

[OMG 97] Object Management Group. *OMG Background Information.* http://www.omg.org/.

[OMG COSS] Object Management Group. *CORBAservices: Common Object Service Specification.* Object Management Group, Inc., Framingham, MA..

[ONTOS] ONTOS. *ONTOS*Integrator - Object/Relational Mapping Concepts.* ONTOS.

[OODB 1] Versant. *Versant Object Database Management System.* Versant Inc., http://www.versant.com/.

[OODB 2] Objectivity. *Objectivity Technical Overview Version 4.* Objectivity Inc., http://www.objectivity.com/.

[OODB 3] Object Design. *Objectstore.* Object Design Inc., http://www.odi.com/.

[Orbix] Iona Technologies. *Orbix Object Request Broker.* Iona Technologies, http://www.iona.com/.

[Orfali 94] R.Orfali, D.Harkey. *Client/Server Survival Guide.* Van Nostrand Reinhold. 1994.

[Orfali 96] R.Orfali, D.Harkey, J.Edwards. *The Essential Distributed Objects Survival Guide.* Wiley, 1996.

[Orfali 97] R.Orfali, D.Harkey. *Client/Server Programming with Java and CORBA.* Wiley, 1997.

[OVA 97] IONA Technologies. *Orbix+Versant Adaptor,* IONA Technologies Ltd., Dublin, Ireland, 1997.

[Owen 93] J.Owen. *STEP: An Introduction.* Information Geometers Ltd, 1993.

[POSC 92] Petrotechnical Open Software Coporation. *Technical Program Overview,* POSC Document TR-1,1992.

[POSC 95] Petrotechnical Open Software Corporation. *Software Integration Platform Specification, Epicentre Data Model, Version 2.0.* POSC, Houston, Texas, October 1995.

[POSC 95b] Petrotechnical Open Software Corporation. *Data Access and Exchange Version 2.0 for Epicentre Logical Model.* POSC, Houston, Texas, March 1995.

[Prism 98] PrismTech. PrismTech *Mapping Manager Project.* http://www.prismtech.co.uk/products/.

[POSIX 97] B.R.Butenhof. *Programming with POSIX Threads.* Addison Wesley, 1997.

[Prism 98b] PrismTech. *The EXPRESSIVE Mapping Language (Version 1.1)* Specification. PrismTech Ltd., Sept 1998.

[Prism 99] PrismTech. *OpenFusion CORBA Services - A White paper.* PrismTech Ltd., March 1999.

[PrismTech] http://www.prismtech.co.uk/

[PSS 98] Objectivity Inc., Secant Tech. Inc., SUN Microsystems Inc.. *Persistent State Service 2.0.* OMG Document orbos/98-12-01, Dec. 1998.

[PSS 98b] Inprise Corp., Persistence Software Inc.. *Persistent State Service 2.0.* OMG Document orbos/98-12-11, Dec. 1998.

[PSS RFP] Object Management Group. *Persistent State Service 2.0 Version of RFP.* OMG Document orbos/97-06-07.

[Reverbel 97] F.C.R.Reverbel, A.B.Maccabe. *Making CORBA Objects Persistent: the Object Database Adaptor Approach.* USENIX Association, Conf. on Object-Oriented Technologies and Systems, June 1997.

[Rieken 92] B.Rieken, L.Weiman. *Adventures in UNIX Network Applications*

Programming. Wiley, NY, USA, 1992.

[Rumb 91] J.Rumbaugh, M.Blaha, W.Premerlani,, F.Eddy, W.Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Sauder 97] D.A.Sauder. *An Implementation of the Standard Data Access Interface Using an Object-Oriented Database and a Distributed Object System(CORBA).* STEP project, NIST, 1997.

[Sessions 96] Roger Sessions. *Object Persistence - Beyond Object-Oriented Databases.* Prentice Hall PTR, 1996

[Shan 98] Y.Shan, R.H.Earle. *Enterprise Computing with Objects - from Client/Server Environments to the Internet.* Addison Wesley, 1998.

[Siegel 96] J.Siegel. *CORBA Fundamentals and Programming.* Wiley, 1996.

[Sloman 87] M.Sloman, J.Kramer. *Distributed Systems and Computer Networks.* Prentice Hall, 1987.

[Sol Man] SUN Microsystems. *Solaris 2.5 Manual Pages.* Sun Microsystems Inc..

[Somvil 89] I.Sommerville. *Software Engineering.* Addison Wesley, 1989.

[STEP 94] International Standards Organization. *STandard for the Exchange of Product model data, STEP.* ISO Standard 10303, 1994.

[STEP SDAI] ISO 10303, *Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 22.* Implementation Methods: Standard Data Access Interface.

[STEP SDAIb] ISO 10303, *Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 23.* Implementation Methods: C++ programming language binding to the Standard Data Access Interface.

[STEP SDAIc] ISO 10303, *Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 26.* Implementation Methods: Interface Definition Language binding to the Standard Data Access Interface.

[SUN 97] Sun Microsystems. *JDBC: A Java SQL API Version 1.1.* G.Hamilton, R.Cattell, Sun Microsystems Inc., Jan 1997.

[SUN 97b] Sun Microsystems. *Java Native Interface Specification.* Sun Microsystems Inc., May 1997.

[SUN 98] Sun Microsystems. *Enterprise JavaBeans Technology - Server Component Model for the Java Platform (White paper).*

[Stevens 90] R.Stevens. UNIX Network Programming. Prentice Hall, 1990.

[Strou 91] B.Stroustup. *The C++ Programming Language, second edition.* Addison Wesley, 1991.

[Vadaparty 95] K.Vadaparty. *Persistent pointer: 1. Journal of Object-Oriented Programming,* July-Aug. 1995.

[Vander 93] R.F.Vander Lans. *Introduction to SQL.* Addison Wesley, 1993.

[Vasu 94] V.Vasudevan, R.Anthony. *Approaches for the Integration of CORBA with OODBs.* Motorola SATCOM, Aug. 1994.

[Vinoski 96] S.Vinoski. *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.* IEEE Communications Magazine, Vol. 14, No. 2, Feb. 1997.

[Visibroker] Inprise. *Visibroker Object Request Broker.* Inprise Corp., http://www.inprise.com/.

[Winblad 90] A.L.Winblad, S.D.Edwards, D.R.King. *Object-Oriented Software.* Addison Wesley, 1990.

[Yang 95] Y.Yang. *STEP Application Protocol Implementation.* P.D.I.T., Oct. 1995. http://www.pdit.com/.