

813-RD-014-001

## **EOSDIS Core System Project**

# **Communications Subsystem CORBA Prototype Results Phase One for the ECS Project**

January 1996

Hughes Information Technology Systems  
Upper Marlboro, Maryland

**Communications Subsystem  
CORBA Prototype Results  
Phase One  
ECS Project**

**January 1996**

Prepared Under Contract NAS5-60000  
CDRL Item #

**APPROVED BY**

<u>Naveen Hota /s/</u>	<u>2/1/96</u>
Naveen Hota, CSS Manager EOSDIS Core System Project	Date

<u>Parag Ambardekar /s/</u>	<u>2/1/96</u>
Parag Ambardekar, ECS Rel. A Manager EOSDIS Core System Project	Date

**Hughes Information Technology Systems**  
Upper Marlboro, Maryland

This page intentionally left blank.

# Preface

---

This document is a contract deliverable with an approval code of 3. This document is delivered to NASA for information only, but is subject to approval as meeting contractual requirements.

Any questions should be addressed to:

Data Management Office  
The ECS Project Office  
Hughes Information Technology Systems  
1616 McCormick Drive  
Upper Marlboro, MD 20774-5372

This page intentionally left blank.

# Abstract

---

The Communications Subsystem (CSS) provides the overall communications infrastructure, and the communications services to support other subsystems in the Science and Communications Development Office (SCDO) and the Flight Operations Segment (FOS). This document describes the first phase of CORBA prototyping at CSS. In addition to tracking development of several most significant commercial ORB products, this prototype has brought about a set of application implementations using ORB and object services includes *naming, event, relationships, life cycle, persistence, delegation, and property*. The evaluation package is SunSoft's CORBA1.2-compliant DOE (beta version).

**Keywords:** CSMS, CSS, OMG, CORBA, ORB, CORBAservices, Naming, Event, Relationship, Life Cycle, Persistence, Delegation, Property, DCE, OODCE, Release A.

This page intentionally left blank.

# Change Information Page

<b>List of Effective Pages</b>			
<b>Page Number</b>		<b>Issue</b>	
Title		Original	
iii through x		Original	
1-1 through 1-4		Original	
2-1 and 2-2		Original	
3-1 through 3-6		Original	
4-1 through 4-6		Original	
5-1 and 5-2		Original	
A-1 and A-2		Original	
AB-1		Original	
<b>Document History</b>			
<b>Document Number</b>	<b>Status/Issue</b>	<b>Publication Date</b>	<b>CCR Number</b>
813-RD-014-001	Original	January 1996	

This page intentionally left blank.

# Contents

---

## Preface

## Abstract

## 1. Introduction

1.1	DCE and ECS Project .....	1-1
1.2	Why CORBA .....	1-1

## 2. Objectives and Approaches

2.1	Objectives.....	2-1
2.2	Approaches.....	2-1

## 3. CORBA Commercial ORBs

3.1	Digital's ObjectBroker.....	3-3
3.2	Expersoft's PowerBroker.....	3-3
3.3	HP's ORB Plus .....	3-4
3.4	IBM's SOM .....	3-5
3.5	IONA's Orbix .....	3-6
3.6	SunSoft's NEO (not an acronym).....	3-7

## **4 Application Development Using DOE**

4.2	The Developed Applications.....	4-11
4.2.1	Life Cycle.....	4-11
4.2.2	Naming.....	4-11
4.2.3	Events.....	4-12
4.2.4	Relationship .....	4-13
4.2.5	Persistent Storage Manager (PSM).....	4-13
4.2.6	Delegation .....	4-14
4.2.7	Property.....	4-14

## **5 Conclusions and Outlook**

5.1	Conclusions.....	5-15
5.2	Outlook.....	5-16

## **Appendix A. References and On-line Sources**

### **Abbreviations and Acronyms**

# 1. Introduction

---

## 1.1 DCE and ECS Project

ECS Project is relying on DCE for its implementation of the communications infrastructure. The goal is to use the most advanced yet mature technology to build an integrated distributed system that best utilizes networked resources. DCE meets the criteria of ECS by reducing the difficulties in designing and implementing the distributed system, providing proven heterogeneous interoperability, and offering a set of basic services includes *security*, *name service*, *time service*, and *distributed file system* required by robust and secure applications.

## 1.2 Why CORBA

ECS project is under way at a time when distributed *object* computing is gaining increasing momentum, fueled by rapid advances in hardware and growing desire for software industry to build, upgrade, and reuse software as *components* based on *frameworks*. As is known, Objects draw strengths and derive marvels from their intrinsic characteristics: encapsulation, inheritance, and polymorphism. Objects in a distributed environment, or distributed objects, are even more capable, holding the greatest potential for creating, using, and reusing the so-called component software that is language-neutral and platform-independent. We have seen the power brought about by semiconductor IC components in electronics industry. Distributed object computing promises us a similar revolutionary future when software engineering, to a large extent, becomes a matter of *plug-and-play* with high flexibility, reusability, and reliability. This will be especially true for system integration with COTS *components* from various vendors. Within next two years, distributed *object* computing is believed to be ripe and ready for prime-time deployment and usage.

To bring the concept to reality, that allows software components to be assembled according to requirements and then inter-operate on all kinds of machines in heterogeneous distributed environments, an industry standard has to be in place. In fact, Common Object Request Broker Architecture and Specification (CORBA) of the Object Management Group (OMG, formed in 1989) was born in response to this development and need.

CORBA is a specification that supports the Object Management Architecture (OMA) of OMG. The recently updated OMA Reference Model consists of five major parts: (1) The *Object Request Broker* (ORB), which enables objects to transparently make and receive requests and submissions in a distributed environments; (2) *Object Services* (CORBAServices), which is a collection of services (interfaces and objects) that support basic functions for using and implementing objects; (3) *Horizontal Common Facilities* (or *horizontal* facility interfaces), which are a collection of facilities that form a foundation (hence, horizontal), providing general-purpose capabilities useful in many applications regardless of application content; (4) *Domain Facilities* (or *vertical* domain-specific interfaces), each of which is specific to a vertical market or industry in areas such as healthcare, finance, manufacturing, retail sales, interactive

multimedia, and telecommunications. (5) *Application Interfaces* (non-standardized application-specific interfaces), which are specific to particular end-user applications. OMA components (3) and (4) are referred to as CORBAfacilites whose scope extends from user interface, to information management, system management, task management, as well as many vertical market facilities.

CORBA is ambitious and important: First, it is potentially capable of bringing all existing applications including client/server middleware to the ORB, the object bus, because of the fact that any specification of services is always separated from the implementation by means of *Interface Definition Language* (IDL). Secondly, it provides a solid foundation, with the magic of system self-description, for a *component*-based future. CORBA allows intelligent components to discover each other and inter-operate on the object bus, which is made possible by the ingenious self-describing schemes using interface/implementation repository and *typecodes*.

With the adoption of CORBA 2.0 (December 1994), any proprietary ORB can connect with the universe of ORBs by translating object requests to and from *the Internet Inter-ORB Protocol* (IIOP). In addition, CORBA 2.0 specifies DCE as its first of many optional *Environment-Specific Inter-ORB Protocols* (ESIOPs). A CORBA 2.0 demo was actually showcased at the Object World in San Francisco (August 1995), eight months after the standard's adoption.

OMG CORBAservices specifies an extensive set of ORB-related services including eleven currently adopted standards:

1. *Naming Service* allows objects on the ORB to locate other objects by providing ability to bind a name to an object (e.g., object reference) relative to a naming context. Naming contexts are described with graphs, and the graphs can be supported in a distributed, federated fashion. The service implementation can be application specific or based on existing network directories or naming contexts including ISO's X.500, OSF's DCE, and Sun's NIS.
2. *Event Service* supports asynchronous events, event "fan-in", notification "fan-out", and reliable delivery (both push and pull models) by de-coupling event suppliers and consumers through event channels. Event channels are objects and can be chained together; event suppliers and consumers register their interests in certain events with the event channel(s) while knowing nothing of each other. No specific event type is imposed.
3. *Life Cycle Service* defines operations for creating, copying, moving, and deleting objects and supports compound life cycle operations on graphs of related objects (see Relationship Service below).
4. *Persistent Object Service* provides a set of interfaces for storing objects persistently on a variety of storage servers including Object Databases (ODBMSs), relational Databases (RDBMSs), and simple files.
5. *Transaction Service* provides two-phase commit coordination among recoverable components using either flat or nested transactions. It supports interoperability between different programming models (e.g., object and procedural), interoperability over

federated ORBs, both implicit and explicit propagation, and execution of multiple transaction concurrently through use of TP monitor.

6. *Concurrency Control Service* enables multiple clients to coordinate their access to shared resources through use of locks. The locks have different modes to allow flexible conflict resolution.
7. *Relationship Service* provides a way to create dynamic associations (or links) between objects that know nothing of each other. It allows graphs of related objects to be traversed without activating the related objects. The service can be used to enforce referential integrity constraints, to track containment relationships, and for any type of linkage among components.
8. *Externalization Service* provides a standard way for getting data into and out of objects in a stream-like mechanism, and usually works in association with Life Cycle Service and Relationship Service.
9. *Query Service* provides query operations for objects. It's a superset of SQL based on the upcoming SQL3 specification and the Object Database Management Group's (ODMG) Object Query Language (OQL).
10. *Licensing Service* supports any model of usage control at any point in a software component's life cycle. It supports charging per session, per node, per instance creation, and per site.
11. *Properties Service* provides operations to let you associate named-values pairs (or, properties) with an object dynamically.

Five more services are expected to be standardized in 1996, with Time and Security services approved early in the year:

1. *Time Service* keeps clocks on different machines synchronized in the distributed environment.
2. *Security Service* provides support for authentication, authorization, audit trail, and non-repudiation.
3. *Trader Service* provides a kind of directory service that is reminiscent of Yellow Pages (phone book), as opposed to White Pages whose function is fulfilled by the naming service. A trader object stores information such as object reference and type of service whenever a service provider (object) registers with it, and subsequently performs "matchmaking" tasks when it is requested by a client.
4. *Collection Service* allows manipulation of objects in a group, such as queues, stacks, lists, arrays, trees, sets, and bags.
5. Change Management tracks versions and evolution history of individual software components as well as objects' interfaces and implementations.

This page intentionally left blank.

## 2. Objectives and Approaches

---

### 2.1 Objectives

Given the importance of the CORBA technology and the nature of ECS project, it is imperative to continue to follow the latest advances of CORBA technology and to investigate the feasibility of CORBA technology for future ECS releases.

The goal of our CORBA prototyping is as follows: Continue to follow advances of CORBA specifications, CORBA services, and CORBA facilities; continue to track maturation of vendor software in areas such as product performance and service availability. Prototype and validate ECS migration strategy by testing selected services for trail migration (gauge difficulties, costs, etc.). Identify migration paths from OODCE implementation to full object implementation using ORB products.

### 2.2 Approaches

For the first phase of this investigation, the effort has been the evaluation of *Distributed Objects Everywhere* (DOE, beta version) from SunSoft whose formal release became available in October 1995 and was renamed as Solaris NEO. This product is CORBA1.2-compliant and is arguably the first complete development, operating and management environment for object-oriented networked applications.

The first phase of the prototype has gone beyond evaluating DOE demos that came with the product. It is a successful implementation of several CORBA applications. Our aim has been the eventual DCE migration to CORBA. To gain first-hand experience in programming with DOE, each application is centered on utilizing one of the several object services supported by DOE. The services include naming, events, relationship, life cycle, transparent persistence management, delegation, and property.

The second phase of investigation that immediately follows, will be more challenging and promise to be the "holy grail" of the whole endeavor: Identify and test-implement migration from OODCE to CORBA based on the experiences from the first to create CORBA objects that will be able to communicate and interoperate with OODCE objects.

The preliminary strategy is to build necessary wrapper objects that will facilitate interactions among objects from the two camps, CORBA and OODCE. In all, three sets of objects will be built including CORBA objects, OODCE objects, and wrapper objects.

This page intentionally left blank.

## 3. CORBA Commercial ORBs

---

### 3.1 Digital's ObjectBroker

ObjectBroker 2.6 is a CORBA 1.2 compliant product. It is layered on top of either TCP/IP or DECnet. Future releases will support the DCE RPC and Microsoft's COM. Digital will support a queuing system DECmessageQ that provides asynchronous ORB messaging.

ObjectBroker 2.6 runs on OpenVMS VAX, OpenVMS Alpha, Ultrix Risc, Solaris, SunOS, Digital UNIX, AIX, HP-UX 9000/700 and 800, SGI IRIX, Windows NT intel, Windows NT Alpha, Mac system 7 (client only), MS Windows (client only), OS/2, MVS, OS400, Tandem integrity, Tandem Non-Stop. ObjectBroker 2.6 offers the following features in terms of language bindings and object services:

- + Language bindings for C and C++.
- + Naming services is currently supported via Framework Based Environment (FBE); there is a plan to make it a built-in service instead of a FBE add-on. Other object services are also planned, including Life Cycle, Events, Transactions.
- + Kerberos DCE authentication via DCE's Generic Security Services interface (GSSAPI).

As for the interoperability protocols specified by CORBA 2.0, Digital plans to support DCE RPC for ESIOP before implementing support for GIOP on top of IIOP.

Digital's strategy is to develop and sell COM/OLE compatible family of products. These products will interoperate with CORBA. ObjectBroker is thus aligned.

### 3.2 Expersoft's PowerBroker

PowerBroker 4.0 distinguishes itself with four key characteristics:

1. Scalability: All messaging between remote objects is direct, avoiding the bottleneck of a centralized routing agent. PowerBroker distributed daemons interact only on demand, minimizing overall network overhead and system initialization time.
2. Standards-Based: PowerBroker is the first CORBA 2.0 compliant ORB available for deployment. Expersoft authored GIOP and is a key contributor to IIOP.
3. Interoperability: Any component of a software system legacy, client/server or object-based can be configured as a distributed object that exports its interface through surrogates. Desktop programs can communicate directly with each other, and disparate applications are connected creating a completely unified corporate network.

4. Performance: PowerBroker IPC mechanisms are direct and do not make use of non-object oriented intermediate transport layers (e.g., RPCs). PowerBroker's direct management of IPC results in an order of magnitude advantage in messaging when compared against other commercial ORBs. Asynchronous messaging has been made a key element of PowerBroker's core technology.

PowerBroker 4.0 runs on Solaris, SunOS, HP-UX, AIX, IRIX, Digital UNIX for Alpha, MS Windows (clients only), MS Windows/NT. In addition, it offers the following features:

Three programming models

- + a CORBA compliant environment with C++ language mappings,
- + a CORBA compliant environment with SmallTalk language mappings,
- + a C++ centric programming model;

A set of integrated services

- + Object Naming Service, supports recursive searches, security, events registration
- + Replicated, Federated Naming Services, provides added reliability and efficiency
- + Object Storage Service, provides automatic generation of methods to store and load objects
- + Object Life Cycle Service, automatically generate methods to migrate and duplicate objects
- + Event Service/Publish and Subscribe, provide filters and persistency to events streams
- + Expersoft PowerBroker CORBA/OLE, extends OLE applications into distributed enterprise
- + Expersoft PowerBroker Rogue Wave, extends RW class libraries into distributed objects
- + Expersoft PowerBroker Object Store, integrates ODBMS with the ORB

A set of management tools

- + Domain Tools, dynamically displays the status of processes, objects, and connections
- + Name Tool, a graphical monitoring and management program for the naming services
- + Registration Tool, used for updating domains with new versions of object implementation
- + Domain Management APIs, for building applications that track/initiate management events
- + SNMP, SNMP based tools gain access to environment via a PowerBroker supplied agent.

### 3.3 HP's ORB Plus

ORB Plus 1.0 from Hewlett-Packard was a CORBA 1.2 ORB that ran only on HP-UX and was structured on top of the DCE RPC and TCP/IP. This product was subsequently removed from the market due to its excessive overhead and problems with its C++ language mapping.

Nevertheless, HP remains influential in the development of the CORBA specification. It is speculated that HP may have delayed making ORB Plus 2.0 available to deliver an "industrial-strength" ORB based on DCE. Consequently, ORB Plus 2.0 is expected to take advantage of DCE security, RPC, and directory services. A full DCE/ESIOP is also expected, complemented by a set of CORBA services including Naming, Events, Life Cycle, Persistence, Transactions, Relationship, Externalization, Licensing, and Query.

As a part of its broader distributed-object framework, HP has released its HP Distributed SmallTalk recently (August, 1995), which incorporates OMG CORBA 2.0 specifications. In addition, it offers CORBA services such as Transaction and Concurrency Control.

### **3.4 IBM's SOM**

System Object Model (SOM) is a software standards developed by IBM to ensure the portability of objects across platforms and development languages. It is an object packaging technology that is language-neutral, platform-independent. On a single machine, SOM provides an object-structured protocol that allows applications to access and use objects and objects definitions regardless of what programming language created them. Distributed SOM (DSOM), on the other hand, supports OMG's CORBA standards. Specifically, SOM 2.1 is based on CORBA 1.2 and provides all CORBA 1.2 ORB functionality. It provides a transport-layer encapsulation framework that supports TCP/IP, IPX/SPX, NetBIOS, and SNA. Under this framework, the SOM ORB has been extended to support DCE.

SOM 2.1 currently runs on Windows 3.X, OS/2, AIX, Macintosh, and MVS and has the additional features as follows:

- + Language bindings for C and C++; SmallTalk may be available from language vendors.
- + Replication Framework, makes copies of a single object available concurrently to multiple clients, and maintains consistency among the copies, with updates to any one copy automatically reflected in all other copies.
- + Persistence Framework, allows to save and restore SOM objects to and from a repository.
- + Emitter Framework, produces an output file representing part or all of an object interface definition, making it easy to develop additional language bindings for SOM.
- + Collection Class Framework, gives programmers such frequently needed data structures as lists, queues, and dictionaries, and lets them inherit from and use these SOM classes in applications with no need to re-code or retest the functions.
- + Direct-to-SOM (DTS) compiler support, combined with a DTS compiler, enables development of SOM objects directly from C++ without writing object definitions in IDL. Existing C++ source code can also be compiled to directly produce SOM objects.
- + Metaclass Framework, extends CORBA, allows dynamic creation of components from existing components, and lets you tailor existing components for a given environment by automatically inserting system behavior into binary object classes.

- + Expected in 1996 are CORBA 2.0 support and additional CORBA services including Events, Life Cycle, Transaction, Concurrency Control, and Externalization. SOM 3.X also intends to offer security based on DCE and ORB-based system management based on Tivoli.

SOM 2.1 is an important CORBA ORB, because SOM provides OpenDoc's underlying CORBA-compliant object bus.

### 3.5 IONA's Orbix

Orbix 2.0 is CORBA 2.0 compliant, supports both the Orbix protocol for optimized communication between Orbix objects and the OMG IIOP for communication with objects in other IIOP compatible environments. Orbix uses TCP/IP as its transport layer. It builds on sockets (WinSock on the Windows platform). It also supports the ONC-RPC. Orbix provides a transport-layer encapsulator that shields applications from the underlying transports.

The entire Orbix system is implemented in C++ and is very portable. Orbix 2.0 runs on Solaris 2.x, SunOS 4.1.x, HP-UX 9.x, IRIX 5.x, AIX 3.2.5, DEC Alpha OSF/1 2.0, DEC Ultrix 4.3, Novell UnixWare 2.0, Windows NT 3.5, Windows 3.1. It has additional features as follows:

- + New C++ mapping in full compliance with the OMG specification; Ada and SmallTalk mappings are expected in early 1996
- + Naming, Event CORBA services. (Transaction Service is expected in early 1996).

Orbix's extensions to CORBA:

- + Implementation repository and administration tools The repository is used to locate the executable files for a server when a request arrives for one of its objects.
- + A stream based interface to the DII C++ class libraries that encapsulate CORBA DII are provided, which makes writing DII clients simpler using familiar stream-like calls.
- + Programmer's control over proxies/surrogates Proxies are local representatives for remote objects. In performance-sensitive applications, server programmers can override the standard proxy code (using inheritance) and implement strategies to cache state and accept callbacks from server objects, which is also referred to as smart proxy.
- + Collocation of client and server code Client and server code can be linked together in the same address space, without requiring recompilation. The resultant code is highly efficient (bypassing all marshaling stubs).
- + Process level filters Programmers can develop their own filter code for incoming and outgoing messages for both clients and servers. This facilitates integration of thread packages, monitoring and debugging, auditing and authentication/authorization/encryption support.
- + Object level filters Programmers can develop their own filter code for invocations and responses on individual objects. This is frequently useful to enable a group of associated objects to collectively respond to a request.

- + Location Service A consultation service that assists the binding of client object references to remote servers, when the name of a host providing the service is unknown.
- + Loaders and object fault handling For servers with a large number of objects, it might be impractical to hold all of the objects in memory. When an invocation on an object arrives at its server, application specific "loader" code can be used to load the object from a file or other storage, and resume the invocation transparently to the client.

IONA is partly owned by SunSoft and is ready to bridge Solaris NEO with OLE to provide interoperability between the two environments. In addition, IONA is aligned with ODI to make Orbix objects persistent by storing them in ODI ObjectStore and to make ObjectStore objects accessible remotely by making them CORBA compliant. IONA is also working on integrating Orbix with Novell's Tuxedo TP Monitor, which will provide process management and load balancing for Orbix objects.

### 3.6 SunSoft's NEO (not an acronym)

NEO, formerly known as Distributed Object Everywhere (DOE), consists of **Solaris NEO**, the networked object add-on extension to the Solaris operating environment; **Solstice NEO**, object administration tools bundled with Solaris NEO; and **WorkShop NEO**, the software product suite for networked object application development and the tools needed to program. NEO is based on OMG CORBA and OpenStep standards. Currently it is CORBA 1.2 compliant. SunSoft plans to have the core features of CORBA 2.0 shipping in the fourth quarter of 1996.

Solaris NEO is unique in that it is an integral part of the Solaris operating environment, built on Solaris' strengths such as robustness and multithreading, and is not merely a middleware ORB implementation as are most CORBA-compliant products. It will interoperate with ORBs on other platforms rather than be ported. Interoperability and application portability are to be achieved through partnerships with other vendors (e.g., IONA for the OLE/COM to CORBA interoperability), support of CORBA2, and Java for Internet access.

Solaris NEO uses TCP/IP as its underlying protocol. NEO IDL compiler (OMG-compliant) supports C and C++ language binding. By adding support for the Java language to the IDL compiler, SunSoft is working on integrating the Java (language) and HotJava (Browser) technology with NEO's object network to facilitate Web-based front-end applications. This technology has been nicknamed *JOE*.

Solaris NEO object services include Naming, Events, Life Cycle, Relationship, Property, Persistence and Delegation. Additional services including Transaction, Concurrency Control, Externalization, Licensing, and Query will be provided in the next release. The big advantages that NEO offers are:

- + Object administration and management
- + Network scalability through MT/MP and network object services
- + Windows/OLE interoperability on the PC's with NEO (via IONA's software)
- + Access to legacy data including RDBMS access (via Persistence Software)

- + Comprehensive development environment for building networked objects and assembling those networked object into applications
- + A CORBA-compliant implementation of OpenStep
- + Internet/Web support (Java)

Additionally, OpenStep is the richest, most elegant open systems GUI development and deployment environment on the market today. The NEO OpenStep API and development tools will provide the most advanced GUI environment for distributed object applications. At the same time, developers may choose to develop Motif/CDE applications.

The Solaris NEO network has SNMP traps built into it, so Solstice and other SNMP tools can be used to monitor it.

# 4 Application Development Using DOE

---

## 4.1 Development Overview

DOE provides a broad set of distributed application tools and infrastructures for developers. The high-level components are briefly described below.

- + **Object Development Framework (ODF)** DOE's development infrastructure that makes developing ORB objects easy. DOE ODF contains components that:
  - + Define an object implementation's characteristics in an implementation file This allows a developer to declare server characteristics. These include concurrency control (locking policy), object creation operations, object installation details and persistent data management. The ODF then generates code based on this characterization.  
*.impl* a notation used to referred to, as well as a suffix to, the implementation file.
  - + Define persistent data using the Data Definition Language (DDL) in a data definition file DDL language allows a developer to define the persistent object data for an ORB object. ODF's persistent data storage mechanism uses this definition to transparently read and write persistent ORB object data.  
*.ddl* a notation used to referred to, as well as a suffix to, the data definition file.
  - + Provides facilities for tracing and logging ORB object status information.
  - + Provides additional runtime ease-of-use functions.
- + **Interface Support** Support for the Interface Definition Language (IDL), as defined in an interface definition file, implements two powerful features of DOE: transparent distributed communication across the ORB and client/server language independence.  
*.idl* a notation used to referred to, as well as a suffix to, the interface definition file.
- + **DOE Debugger** In addition to traditional debugger features, the DOE Debugger allows a developer to trace code from a distributed client into the object server code and back again.
- + **CORBA services** DOE provides a basic set of CORBA services that implement common tasks such as naming, relationships, property, and event management.

In the course of the first prototyping phase, each application has been developed in three stages: (1) creation and compilation of IDL; (2) implementation and compilation on the server side; (3) implementation and compilation on the client side. Each stage is organized in a separate sub-directory, due to the many files generated in each process.

To develop a new object server, follow these steps:

1. Define a *.idl* file describing the interface for the objects that is being implemented Define the interfaces to be supported by the object server in one or more *.idl* files. Later, provide separate C++ implementation code to support these interfaces. IDL interfaces are language/implementation independent. This allows a developer to modify object implementation over time without compromising the plug-and-play qualities of the object.
2. Define a *.impl* file describing the object implementations The implementation definition file allows a developer to customize how the objects are to be created, activated and deactivated, when the server may timeout, and other useful runtime options. The file is processed by the *odfcc* compiler to produce C++ code that the developer would otherwise need to write himself.
3. Define the objects' DDL if ODF managed persistence is to be used ODF will automatically and transparently save any data members declared in a DDL (Data Definition Language) file. A developer can also create or use his own persistence mechanism. It is strongly recommended that all objects be made persistent.
4. Create *odfmake* Imake macros are used for defining ODF based servers, ODF based clients, IDL interface libraries, ODF implementation binaries, DDL schema binaries, shell scripts, exception message catalogs, user started servers, and packages. These macros have to be organized and adapted to each application.
5. Run *odfmake* This command uses the Imakefile composed with Imake macros to generate a standard ODF Makefile.
6. Run *make impl* The *odfcc* compiler will generate C++ implementation "skeletons".
7. Run *make copy\_samples* This step copies the skeleton files to working directories from *odf\_output* subdirectories.
8. Define the C++ implementation classes in the header *.hh* file and write method implementation code in the *.cc* file.
9. Build the object server Type *make* with no arguments to build the IDL stub library, DDL type files (if applicable), and the object server. Each server may run one or more objects.
10. Register the interface and server with the ORB and create factory objects and register them in the Naming service Type *make register*.

The following section provides a brief description for each application set developed in the first phase of this prototype. Contact the author for in-depth design and implementation issues. A demo can also be arranged.

## 4.2 The Developed Applications

### 4.2.1 Life Cycle

DOE-provided lifecycle interfaces are used to create and destroy an object; interfaces for move and copy are not supported in DOE 1.0.

In this application, the primary goal is to take advantage of the *hooks* that are called during the life cycle of a server, an object implementation, and an object instance, to manage own persistence (instead of using the system provided Persistence Storage Manager, Sec. 4.2.5). The idea is to implement our own mechanism to keep the object persistent, by means of saving the data member(s) of an object into a file (or a database) when the object deactivates, and retrieving the data back when the same object is activated next time. The data member(s) is copied into a data structure before this data structure itself is written to the file with an index. The index records offset of the structure within the file. Each structure also carries a flag indicating whether the file space occupied by the structure is still in use or it is no longer in use and ready to be reused.

A name is assigned and stored together with the index (or the offset indicating the structure's location within the file) as a name-value pair in the object's reference data store. All DOE objects are given a small reference data store of size 1024 bytes when it is initially created.

The following are the hooks that correspond to different lifecycle events and are specified in the implementation file *.impl*:

1. Registration Create a data file when the server is registered; this happens when *make register* is executed to register the interface and the server. The directory in which the file will be created, can be specified in the *Imakefile* to override default.
2. Server Startup Open the file and store the file descriptor.
3. Object Initialization Obtain and store the object reference of a newly created object in a name space.
4. Object Activation Read the instance's data from the persistent data file.
5. Object Deactivation Write the instance's data to the persistence data file.
6. Server Shutdown Close the file that has been opened when the object activated.

Other task-specific operations are specified in the *.idl* file (interface definition) and are invoked explicitly by a client program rather than operate as *hooks* at the server side.

### 4.2.2 Naming

This application consists of three client programs: *listContext*, *copyContext*, and *delContext*. The provided naming service is used by including the *Naming.idl* file in the application's *.idl* file. *Naming.idl* has defined a rich set of interfaces that are CORBA-compliant; this application uses only a selected set of them.

1. *listContext*, acquires object reference from a specified naming context and list all the name-to-object-reference bindings contained within the naming context. This is similar to "ls" command in Unix.

To accomplish this task, build an initial context and a *CosName* representation for the path which is a naming context specified as a command line argument; obtain the object reference to the naming context (an object by itself); obtain and iterate over a list of bindings in the context, printing each one.

2. *copyContext*, create a new naming context, copy another context's bindings into it, and bind it within a third context so that it may be found. This is similar to "cp" in Unix.

To accomplish this task, first traverse to a specified source naming context as in the implementation of *listContext*, then iterate through the name bindings it contains and duplicate them in the destination naming context which is then bound to the initial naming context.

3. *delContext*, destroys the naming context created previously in *copyContext*. This is similar to "rm" in Unix.

To accomplish this, traverse to the naming context to be destroyed, unbind its name bindings, and then destroy the naming context.

### 4.2.3 Events

This application consists of five cooperating sub-applications conducting asynchronous communications among the participants. The provided event service is used to configure the "event channel" by adding/removing suppliers and/or consumers, and to perform event "fan-in" and notification "fan out" through the event channel.

1. *createEventChannel* This is a client program that creates an event channel and register it in the name space. The created event channel will be used by the following four applications.
2. *pullConsumer* This is a client program that pulls information from the event channel.
3. *pushSupplier* This is a client program that prompts users for information, and then pushes it into the event channel for distribution to any registered consumers.
4. *pushConsumer* This is a server program implementing the *CosEventComm::PushConsumer* interface. The server will print out any information it receives (pushed to it from the event channel). A separate client program is used to create the actual push consumer.
5. *pullSupplier* This is a server program that is called upon by a separate client program to create an event item (and, signal "more"). The server implements its own buffering using a structure array. At any time, the event channel may pull an item out of the buffer (and, signal "less") if the buffer is not empty. A "producer/consumer" algorithm is used. In a two-threads scenario: One thread, the mt-producer, writes items into the buffer if space is available, else it waits. The other thread, the mt-consumer, reads/removes items from the

buffer when they are available, else it waits. To avoid blocking too many threads, the object function raises an exception if the buffer is full rather than blocks.

ODF provides one mutex lock per object by default; any given object only services one client request at a time. To be able to implement the MT producer/consumer algorithm specifically, using `cond_wait(cond_t *, mutex_t *)` and `cond_signal(cond_t *)` to coordinate the usage of the self-managed buffer a finer-grained locking mechanism is used so that each condition variable is tested under the protection of the same mutex lock as the one used for locking an individual method.

#### 4.2.4 Relationship

This application consists of four parts, which builds relationships at run time among objects. The relationships can be traversed starting from an arbitrary object.

1. *Folder* Create an otherwise normal object, *Folder*, in the server program. To allow itself to be able to participate in a Containment relationship, *Folder* inherits "NodeDelegate" in its interface from the Relationships.idl supported by the relationship service and adds to itself a "contains\_role" at its initialization time.
2. *Document* Create an otherwise normal objects, *Document*, in another server program. To allow itself to be able to participate in a Containment relationship, *Document* inherits "NodeDelegate" in its interface from the Relationships.idl supported by the relationship service and adds to itself a "contained\_in\_role" at its initialization time.
3. *CreateRelationship* A client program finds objects including the two documents, folder, and ContainmentRelationshipFactory from the name space; then uses `roles_of_type` operation to get respective roles of documents and folder objects; finally builds two Containment relationships. The involved folder and documents objects have no knowledge of the relationships being built. The relationships are objects themselves; the same are true for roles and nodes that participate in a relationship.
4. *List* This is a client program that takes advantage of the existing relationships and navigates among the distributed objects that are otherwise independent. Setting out from any one object, the client hops from one object to another and asks each one to perform certain tasks, in this case, listing each object's name and the role it plays in the relationship.

#### 4.2.5 Persistent Storage Manager (PSM)

Unlike the self-managed persistence in Life Cycle (Sec. 4.2.1), PSM is a service provided by DOE to make things easier. This application is ingenuous and thus a good starting point programming with DOE. In this application, an "adder" object is created. The adder takes two integers from a client and performs the addition before returning a result. Remember, all this happens over the network; the object may deactivate, and the server may shutdown. By using the support of PSM, the "adder" object is able to recover, hence the client program is able to obtain the adder's result that has been calculated prior to its deactivation.

Persistent attribute(s) is declared in the `.ddl` file; PSM is specified in the `.impl` file.

## 4.2.6 Delegation

Like persistence management, implementation reuse can be realized by means of either user-implemented delegation or ODF's support for delegation. Delegation, in this context, means that one object relays a method invocation to another object which will either complete the task or relay the invocation one step further until the task is done.

Re-using the adder's implementation, this application implements a slightly more sophisticated object, "adder-deductor", that is capable of subtraction as well as addition using the ODF's support (in fact, a user-implemented delegation has also been successfully programmed). The *.idl* file incorporates adder's operation through interface inheritance and defines its own subtraction operation. Addition method invocation is delegated transparently, while subtraction method invocation is passed to the adder after reversing sign of the second argument.

Adder object is specified as a delegate in the *.impl* file.

## 4.2.7 Property

One way to add attributes to an ORB object is through the IDL interface, in which case IDL attributes are statically fixed at compile-time. By contrast, properties are dynamic attributes which may be added and removed at run-time and without any involvement on the object's part similar to how relationships among objects are built. In that application (Sec. 4.2.4), a folder contains two documents; each document is contained in a folder. Suppose at one point, for example, we want to archive all the documents and folders on the network and need to assign each of them an ISBN number, an attribute that the folder and document objects do not have at compile-time. How can we accomplish this? Property service provides a solution.

This application uses property service and is based on the previous Section to build a primitive distributed calculator object. This object is able to conduct operations such as addition, subtraction, multiplication, and division. The first two operations are delegated to adder-deductor (Sec. 4.2.6) which are then delegated further to the adder object (Sec. 4.2.5). Properties describing functionality of the calculator is supported by the PropertySet object through a separate delegation. PropertySet is an interface within PropertyService and is part of the calculator object's interface through interface inheritance.

Two client programs have also been implemented: One adds properties, or attributes, to the calculator object; the other actually makes use of that information by accessing these attributes.

## 5 Conclusions and Outlook

---

### 5.1 Conclusions

DOE has proven itself to be a viable ORB product and is relatively easy to program with (as compared to DCE). The extensive CORBA services have made building a modestly sophisticated set of objects an easy task.

Overhead varies for method invocation on network objects, depending on how many services (or, server objects) are active. It may take longer time for the invoked object to respond if required services are deactivated. The response time, however, approaches that of a local application if all required services are active and available.

A distributed object becomes deactivated, and a server shutdown if either has not been used for a specific period of time, so as to save network resources. A server has to be started up before an object can be activated. On the other hand, objects within a server have to be deactivated before the server can be shutdown. The duration for objects and servers to remain active is configurable at server installation.

Table 5-1 provides an *estimated* range between the longest response time when server is down and the shortest response time when involved objects are active and available. This information should be used as a guide and by no means applicable under all circumstances, due to differences in hardware and testing environment (e.g., how many processes are running concurrently).

**Table 5-1. Object Service vs. Response Time**

Object Service	Response Time* (seconds)	Response Time* (seconds)
Naming Service	16	1
Events Service	11	1
Relationship Service	14	1
Persistent Storage Manager	3	1
ODF Delegation Service	3	1
Property Service	10	1

\* The recorded figures are the elapsed (wall clock) time; the CPU time consumed by each individual service is less than 0.1 seconds.

The machine in use was a SPARCstation 10 (Sun 4/80), having 64 MB of RAM. Currently, no authoritative bench mark for ORB products is available, but some are said to be in production or planned.

## 5.2 Outlook

Future tasks for the application of CORBA-compliant products should be two-fold. First, continue to carry out OODCE to CORBA migration, beginning with prototype, as this is relevant to ECS project. Secondly, evaluate ORBs from other vendors, possibly including IONA, IBM, HP, ExperSoft, Digital, as well as SunSoft. This should go beyond the association with DCE, because CORBA and DCE offer technologies that are quite different.

It should be noted that an ORB product that supports DCE IOP (CORBA2 ESIOP) does not necessarily make itself a better candidate for future releases of ECS. From the viewpoint of system integration, there is no instant interoperability between ORB objects and OODCE objects, regardless of the type of inter-ORB protocol supported by the ORB product.

## Appendix A. References and On-line Sources

---

### **OMG Standards:**

CORBA: Architecture and Specification 2.0 (July, 1995), OMG.

CORBAservices (March, 1995), OMG.

CORBAfacilities (November, 1995), OMG.

Vendor newsletters.

### **On-line Sources:**

World Wide Web Site: <http://www.omg.org>

FTP Site: <ftp.omg.org>

Internet Address: [info@omg.org](mailto:info@omg.org)

Vendor WWW Home Pages.

For specific OMG documents, go to OMG FTP site (see above) where a rich source of information can be found. File */pub/docs/doclist.txt* contains name-and-sequence-number indices for all the documents in the directory */pub/docs*.

This page intentionally left blank.

# Abbreviations and Acronyms

---

COM	Component Object Model (Microsoft), or Common Object Model (Digital)
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-the-Shelf (hardware or software)
CSMS	Communications and System Management Segment
CSS	Communications Subsystem
DCE	Distributed Computing Environment
DDL	Data Definition Language
DOE	Distributed Object Everywhere
DSOM	Distributed SOM
ECS	EOSDIS Core System
EOS	Earth Observing System
EOSDIS	EOS Data and Information System
ESIOP	Environment-Specific Inter-ORB Protocol
FBE	Framework Based Environment
GIOP	General Inter-ORB Protocol
IC	Integrated Circuit
IDL	Interface Definition Language
IIOB	Internet Inter-ORB Protocol
ODBMS	Object Database Management System
ODF	Object Development Framework
OLE	Object Linking and Embedding
OMA	Object Management Architecture
OMG	Object Management Group
OODCE	Object-oriented DCE
ORB	Object Request Broker
OSF	Open Systems Foundation
PSM	Persistent Storage Manager
RDBMS	Relational Database Management System
RPC	Remote Procedure Call
SNMP	Simple Network Management Protocol
SOM	System Object Model