2002

Migration of legacy DCE systems

Daniel J. Jensen
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Jensen, Daniel J., "Migration of legacy DCE systems" (2002). *Master's Theses*. 2278.
DOI: <https://doi.org/10.31979/etd.g4qm-b2hs>
https://scholarworks.sjsu.edu/etd_theses/2278

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

MIGRATION OF LEGACY DCE SYSTEMS

A Thesis

Presented to

The Faculty of the Department of Engineering

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Daniel J. Jensen

May 2002

UMI Number: 1408795

UMI Microform 1408795

**Copyright 2002 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

©2001

Daniel J. Jensen

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF GENERAL ENGINEERING

RA Fatoohi 12-13-2001

Dr. Rod Fatoohi

Dan Harkey 12-12-2001

Prof. Dan Harkey

J Dorosti 12/12/01

Dr. Javad "Jim" Dorosti

Jim Dorosti

APPROVED FOR THE UNIVERSITY

ABSTRACT

Migration of Legacy DCE Systems

by Daniel J. Jensen

Legacy applications based on the *Distributed Computing Environment* (DCE) are subject to several major limitations. DCE is no longer in development. It is not maintained and is largely unsupported. DCE does not support object-oriented development, languages other than C, or transmission over HTTP. These shortcomings press DCE application managers and developers to find extensions and alternatives to DCE.

This thesis discusses several plausible targets for migration and extension of DCE systems, then proceeds to detail, compare, and contrast two preferred candidates: the *Common Object Request Broker Architecture* (CORBA) and the *Simple Object Access Protocol* (SOAP).

This thesis explains that developing a general migration solution for DCE applications to CORBA or SOAP would be difficult, however, it would be feasible to develop targeted solutions for systems that depend on a large sub-domain of the DCE feature set. Given a short list of reasonable premises, many DCE applications may be ported to CORBA or SOAP.

ACKNOWLEDGEMENTS

A portion of the research performed in support of this thesis has been funded by the National Aeronautics and Space Administration (NASA) through cooperative agreement number NCC2-1071 with the San Jose State University Foundation. The purpose of this cooperative agreement has been to fund research into middleware alternatives for NASA's Earth Observing System (EOS) Data and Information System (EOSDIS).

My loving thanks to my wife Carolyn and daughter Brenna for their patient support.

TABLE OF CONTENTS

1	INTRODUCTION.....	1
1.1	OBJECTIVE	1
1.2	CASE BACKGROUND: OODCE AND EOSDIS.....	1
1.3	TECHNOLOGY BACKGROUND: DCE AND OODCE	3
1.3.1	<i>OSF/DCE</i>	3
1.3.2	<i>Challenges Confronting DCE Applications</i>	4
1.3.3	<i>Emerging Alternatives to DCE</i>	6
1.4	CANDIDATES FROM THE MIDDLEWARE MARKETPLACE	8
1.4.1	<i>ONC RPC</i>	8
1.4.2	<i>OSF DCE</i>	9
1.4.3	<i>DCOM</i>	11
1.4.4	<i>CORBA</i>	13
1.4.5	<i>SOAP</i>	18
1.4.6	<i>Summary</i>	20
1.5	MIGRATION AND INTEGRATION OVERVIEW	21
2	PRINCIPLES OF MIGRATION	25
2.1	LEVELS AT WHICH MIGRATION OCCURS.....	25
2.2	ADAPTERS AND BRIDGES	26
2.3	ACHIEVING SOURCE-LEVEL INTEROPERABILITY	28
2.4	LIMITATIONS TO INTEROPERABILITY	29
2.4.1	<i>Special DCE RPC Features</i>	29
2.4.2	<i>Supplementary Services</i>	32
2.4.3	<i>Non-RPC Services</i>	34
3	MIGRATING TO SUN/ONC RPC	35
3.1	INTRODUCTION: FEATURE SUPPORT	35
3.2	INTERFACE COMPATIBILITY	39
3.2.1	<i>Interface Definitions</i>	39
3.2.2	<i>Application Programming Interfaces</i>	39

3.3	OODCE-OVER-ONC	44
3.4	CONCLUSIONS	44
4	MIGRATING TO DCOM	46
4.1	GENERAL DISCUSSION	46
4.2	WRITING DCE PROGRAMS FOR DCOM	46
4.2.1	<i>Analysis</i>	46
4.2.2	<i>Porting a Simple DCE Client/Server Pair to DCOM</i>	50
4.3	OODCE	52
4.4	MOVING TO DCOM INTERFACES	53
4.5	ADDED VALUE	57
4.6	CHALLENGES	59
4.6.1	<i>The Risk of Vendor Lock-In</i>	59
4.6.2	<i>Platform Support</i>	61
4.6.3	<i>Measuring the Impact of Missing Windows Services</i>	62
4.6.4	<i>Recent Enhancements to DCOM Portability</i>	62
4.6.5	<i>A Long View</i>	62
4.6.6	<i>Java Interoperability</i>	65
4.6.7	<i>Scalability</i>	66
4.6.8	<i>The Client</i>	66
4.7	CONCLUSIONS	68
5	MIGRATING TO CORBA	69
5.1	CHALLENGES	69
5.2	SOLUTIONS	71
5.2.1	<i>Bridging</i>	71
5.2.2	<i>Alternatives to Bridging</i>	75
5.2.3	<i>Source-Level Interoperability via Adapters</i>	77
5.2.4	<i>Combining Bridging with Source-Level Interoperability</i>	79
5.3	TECHNICAL DISCUSSION	81
5.3.1	<i>Marshalling Strategies</i>	81
5.4	DESIGN	84
5.4.1	<i>Requirements</i>	84
5.4.2	<i>Method Invocation Flow</i>	85
5.4.3	<i>The Code Migration Process</i>	87

5.4.4	<i>Migration Process Steps</i>	89
5.4.5	<i>Code Generation</i>	91
5.5	CASE SPECIFICS	97
5.5.1	<i>Simple and Threaded Math Examples</i>	97
5.5.2	<i>Simple Implementation of the ECS Subscription Server Interface</i>	98
5.6	PROCESS	101
5.6.1	<i>Simple and Threaded Math Examples</i>	101
5.6.2	<i>The ECS Subscription Server Interface</i>	103
5.7	PREVIOUS RESEARCH	104
5.8	GENERATING AN OMG INTERFACE DEFINITION FILE	109
5.8.1	<i>Feasibility</i>	109
5.8.2	<i>Developing a Conversion Utility</i>	112
5.8.3	<i>The Conversion Utility</i>	112
5.8.4	<i>IDL Elements Not Supported</i>	113
5.9	THE ELEMENTS OF IDL CONVERSION	115
5.10	OTHER ISSUES	117
5.11	THE CLIENT	118
5.11.1	<i>Miscellaneous Details</i>	118
5.11.2	<i>Basic Form of a Inter-Middleware Invocation</i>	120
5.11.3	<i>Advanced Type Mapping</i>	121
5.12	THE SERVER	122
5.13	DCESERVER	124
5.14	OBSERVATIONS	126
5.14.1	<i>Changes in VisiBroker</i>	126
5.14.2	<i>Problems with Skeletons and Stubs</i>	127
5.14.3	<i>Conclusions</i>	129
5.14.4	<i>Minor Problems</i>	132
6	RPC OVER IIOP	133
6.1	INTRODUCTION	133
6.2	RPC-API OVER IIOP	134
6.2.1	<i>DCE</i>	134
6.2.2	<i>OODCE</i>	148
6.2.3	<i>Conclusions</i>	155
6.3	RPC STUBS OVER IIOP	157

7	SIMPLE OBJECT ACCESS PROTOCOL (SOAP)	159
7.1	OBJECTIVE	159
7.2	THE ARCHITECTURE	161
7.2.1	<i>DCE Method Invocation Flow via SOAP</i>	162
7.2.2	<i>SOAP Routing</i>	168
7.2.3	<i>A SOAP Broker Example</i>	171
7.3	DESIGN	173
7.3.1	<i>Code Generation</i>	173
7.3.2	<i>Component Diagrams</i>	178
7.4	CASE SPECIFICS	187
7.5	PROCESS	187
7.5.1	<i>Simple and Threaded Math Examples</i>	189
7.5.2	<i>Subscription Server Example</i>	189
7.6	A CODE WALK THROUGH AN OODCE/XML REMOTE INVOCATION	190
7.6.1	<i>The Path of the Function Invocation Through the Client</i>	190
7.6.2	<i>The Path of the Function Invocation Through the Server</i>	195
7.6.3	<i>The Path of the Function Return through the Server</i>	201
7.6.4	<i>The Path of the Function Return through the Client</i>	205
7.7	IMPLEMENTING HTTP FOR DCE-OVER-SOAP	206
7.7.1	<i>The Multiple Process Dilemma</i>	206
7.7.2	<i>Embedded HTTP facilities</i>	208
7.7.3	<i>Bridging</i>	209
7.8	OBSERVATIONS	211
7.9	OUTCOME	214
8	BUILDING BRIDGES FROM ADAPTERS	215
8.1	A SOAP BRIDGE	215
8.2	A CORBA-DCE BRIDGE ARCHITECTURE	218
9	SUMMARY OF DEVELOPMENT ACTIVITIES	219
10	PERFORMANCE	221
10.1	SUMMARY	221
10.2	DETAILS	223

10.3	OODCE-OVER-SOAP OVERHEAD	225
10.4	BRIDGES.....	226
11	CONCLUSIONS	228
12	DEVELOPMENT TOOLS AND ENVIRONMENT	231
13	CORBA 2.X ORBS WITH C++ BINDINGS.....	232
13.1	COMMERCIAL ORBS.....	232
13.2	NON-COMMERCIAL ORBS	234
14	WORKS CITED.....	235
14.1	RPC	235
14.2	DCE.....	235
14.3	OODCE.....	236
14.4	DCE AND CORBA	236
14.5	CORBA	237
14.6	BRIDGING.....	238
14.7	EOSDIS CORE SYSTEM (ECS)	238
14.8	DCOM.....	239
14.8.1	<i>Windows NT Development</i>	240
14.8.2	<i>DCE Interoperability and Portability</i>	240
14.8.3	<i>Ports of COM/DCOM to UNIX</i>	240
14.9	SIMPLE OBJECT ACCESS PROTOCOL (SOAP).....	241

FIGURES

Figure 1: Migration Targets to be Addressed.....	21
Figure 2: OODCE programs running over CORBA and SOAP	23
Figure 3: Adding SOAP Clients and Servers	24
Figure 4: CORBA Marshalling	81
Figure 5: Dual-Stub DCE/IIOP Marshalling.....	82
Figure 6: DCE Method Invocation Flow via CORBA.....	83
Figure 7: Invocation Flow Diagram	85
Figure 8: Migration Process	87
Figure 9: DCE Method Invocation Flow via SOAP.....	164
Figure 10: DCE Method Invocation Flow via SOAP (detail).....	165
Figure 11: Router-Broker Architecture	169
Figure 12: Multiple SOAP Transports	171
Figure 13: Inter-Process Communication.....	207
Figure 14: Internal HTTP facilities	209
Figure 15: DCE Method Invocation via the SOAP-DCE Bridge.....	217

TABLES

Table 1: OSF and ONC Data Representations	36
Table 2: OSF and ONC RPC Mechanisms	37
Table 3: OSF and ONC Protocol Compilers.....	37
Table 4: OSF and ONC Network Services.....	38
Table 5: OSF RPC Client Library Calls.....	40
Table 6: OSF RPC Server Library Calls	40
Table 7: OSF RPC Binding Calls.....	40
Table 8: OSF RPC Binding Calls (cont.)	41
Table 9: ONC RPC Client Library Calls.....	42
Table 10: ONC Server Library Calls.....	43
Table 11: DCOM Support of the DCE RPC API	48
Table 12: Converter Output.....	89
Table 13: Shared Libraries & Headers	90
Table 14: DCE RPC Stage 1 of 2	137
Table 15: DCE RPC Stage 2 of 2	138
Table 16: OODCE RPC Stage 1 of 4	151
Table 17: OODCE RPC Stage 2 of 4	152
Table 18: OODCE RPC Stage 3 of 4	153
Table 19: OODCE RPC Stage 4 of 4	154
Table 20: Relative Latency of SOAP Transports	222
Table 21: Test Machines	223
Table 22: OODCE-over-SOAP Performance.....	224
Table 23: Overhead	225
Table 24: Bridge Performance	227
Table 25: Commercial C++ ORBS	232
Table 26: Non-Commercial C++ ORBs.....	234

1 Introduction

1.1 Objective

The objective of this thesis is to examine the feasibility of porting legacy DCE/ODCE applications to other middleware platforms, either entirely or in part, with special attention to the *Simple Object Access Protocol* (SOAP) and the *Common Object Request Broker Architecture* (CORBA) as prospective migration targets.

1.2 Case Background: ODCE and EOSDIS

Hewlett-Packard's *Object-Oriented Distributed Computing Environment* (ODCE) product is an extension of the *Open Software Foundation's* (OSF)¹ *Distributed Computing Environment* (DCE).

This thesis uses the *National Aeronautics and Space Administration's* (NASA) *Earth Observing System* (EOS) *Data and Information System* (EOSDIS) as a primary example of a legacy ODCE application.

The EOSDIS project has a number of significant development issues to resolve, not the least of which is its complexity above the middleware layer. EOSDIS is burdened by an inconsistent application programming interface (API). Though the EOSDIS vision was to have a consistent API, expediency undermined this vision as schedules for the components of this API slipped. With this history in mind, this thesis does not propose to cure the ills of EOSDIS by throwing middleware at NASA. Rather, it modestly presents a review of the alternatives for a component of an overall solution which must remain beyond the scope of this investigation.

This discussion continues with an introduction to DCE, the middleware underlying the EOSDIS Core System (ECS).

¹ Presently known as the *Open Group*, but since DCE development predated this new name, the original name *Open Software Foundation* (OSF) is used herein.

1.3 Technology Background: DCE and OODCE

1.3.1 OSF/DCE

DCE, short for *Distributed Computing Environment*, is a product of the *Open Software Foundation (OSF)*.² DCE is based on the *Network Computing System (NCS)* remote procedure call (RPC) based system. NCS RPC is similar to, though not compatible with³, the RPC product developed by Sun Microsystems, formally called ONC RPC.⁴ Both RPC systems have been available on SunOS, HP-UX, and ULTRIX since 1992 at latest⁵.

² The term *DCE* also denotes the general class of distributed computing environments, which includes CORBA and COM.

³ RPC compatibility to be discussed in detail. For a brief summary, see *Power Programming with RPC*, pages 19-20.

⁴ Sun RPC is backed by the industry consortium *Open Network Computing (ONC)*.

⁵ *Power Programming with RPC*, pages 22 and 35.

1.3.2 Challenges Confronting DCE Applications

Several serious challenges are faced by legacy DCE applications.

First, DCE is no longer in development. It is not being enhanced to meet new needs, nor is it being maintained, nor is it formally supported for operation on the most current releases of any operating system. The current release of OSF DCE is 1.2.2. The Open Group Web site specifies that the reference platform for OSF/DCE 1.2.2 is IBM AIX v3.2.5, whereas the current version of AIX is 5.1. Another brand of DCE 1.X, TransArc/IBM DCE 1.1, is not supported beyond Solaris 2.5.1 (Solaris 2.8 is current) [2.4]. There is no 1.2.X version of TransArc/IBM DCE.

DCE does not support object-oriented development. It has been extended by several object-oriented development products, the chief among them having been OODCE. All of these supplementary products have been discontinued. The current DCE FAQ [2.3] (dated October 1998) states that *the Open Group recently announced that OODCE will be adopted as part of DCE in a future version*, but OODCE has been discontinued, leaving customers such as NASA to maintain and support it *in house*.

DCE supports TCP/IP, but it does not support transmission over HTTP, the dominant high-level protocol of today, and a requirement for most firewall products. The OSF sponsored a DCE-Web project in 1995-96 that progressed no further than version 1.0.1. This project is now presented as an example of how DCE services can be made available over the Web, but no products are forthcoming.

Furthermore, DCE provides no bindings to Java (or any language other than C), nor does DCE make any use of the object-oriented features of C++.

1.3.3 Emerging Alternatives to DCE

In recent years, the *Object Management Group's* (OMG) CORBA and Microsoft's DCOM have achieved dominance in the distributed computing market. CORBA dominates the market for truly distributed, heterogeneous application support, whereas COM dominates the market for distributed components and applications based on *Microsoft Windows*.

Other middleware alternatives to DCE have appeared that are based on the Java language and environment: Java RMI and Enterprise Java Beans (EJB). Though both of these technologies are somewhat promising, they are Java-specific. Though RMI-over-IIOP has made it possible for RMI clients to communicate with CORBA servers, there is no way to move legacy DCE clients and servers to RMI without converting code from C/C++ to Java.

Still more recently, The Extended Markup Language (XML) has presented a new formalized representation of data and procedure calls that is achieving rapid acceptance and implementation in industry. The Simple Object Access Protocol (SOAP), an evolving specification for XML-based RPC and messaging, enjoys strong support by industry leaders such as Microsoft and Sun Microsystems. Microsoft is

building its .NET web development framework on SOAP, and Sun Microsystems has made SOAP a core component of their *Open Net Environment* (ONE) initiative [9.17].

1.4 Candidates from the Middleware Marketplace

1.4.1 ONC RPC

Also known as Sun RPC, this is the original cross-platform, openly available, distributed middleware product. As Sun OS (Solaris) has emerged as the dominant UNIX platform, ONC RPC has established itself correspondingly.

The primary issues with moving any application from DCE to this RPC technology are:

- What does one gain by moving to ONC RPC?
- How difficult would it be to migrate to ONC RPC?

This thesis explains that (1) ONC RPC provides little added value to a legacy DCE application, and (2) ONC RPC is not nearly as compatible with DCE RPC as it is functionally similar.

1.4.2 OSF DCE

The option of continuing to use DCE ought to be considered, in the interests of completeness. The DCE/RPC session protocol is far from dead, thanks to DCOM⁶. DCE is not object-oriented in the same sense that CORBA is, but DCE does support an implicit form of object orientation. That is to say that DCE supports the concept of encapsulation, such that procedures can easily be implemented as object methods. This support of object orientation is formalized by OODCE, wherein object orientation is the norm.

DCE without OODCE is a viable path for legacy OODCE applications, as these applications operate on layers over DCE and their IDLs can easily be reused to functionally equivalent procedural stubs. For this reason, the abandonment of OODCE may not necessarily be a fatal event for the few remaining applications that use it.

Of course there are good reasons to pursue migration to advanced distributed object environments such as CORBA and DCOM, but any extensive migration comes at a considerable price.

⁶ Now part of the COM+ product, but this thesis uses the term DCOM where the term COM+, as a superset of DCOM, is superfluous.

This thesis will not pursue this line of discussion any further, because the objective of this thesis is to explore migration alternatives to DCE and OODCE, based upon the observation that DCE has a significant list of shortcomings, rather than exhaustively weighing the strengths and weaknesses of this legacy software against each alternative.

1.4.3 DCOM

DCOM is a very promising, powerful middleware platform that not only uses the DCE RPC wire protocol, but also provides a OSF-compatibility mode and application programming interface that enable DCE code to operate in a DCOM environment with a few very simple source code modification rules. Because of this, much of the DCE development and testing supporting this thesis was performed on *Microsoft Windows* systems.

The question for DCOM as a prospective migration target is *how compatible can a DCOM client, with all of DCOM's additional interface requirements, be with a conventional DCE server?* It is likely that a translation layer would need to be introduced, but that layer would probably be thin, and simple to develop.

DCOM may be regarded as the object-oriented heir to the DCE legacy. The 'D' in DCOM is extracted directly from the DCE variety of RPC. However, DCOM is not an open, cross-platform product. Though much of it has been ported to UNIX recently, parts of DCOM are inextricably tied to Microsoft's *Windows* platform. So it is that, though DCOM is similar in its core RPC technology to DCE, yet in terms of services and APIs, DCOM and DCE systems are incompatible. This narrow focus of DCOM on the *Microsoft Windows* platform has been credited as one of the issues that recently

caused *Microsoft* to shift its focus to its new *.NET* strategy. This should not be seen as an end to DCOM, but rather an acknowledgement that DCOM is not for everybody.

Porting large legacy applications to *Windows*, especially mission-critical applications, is not usually feasible. This appears to be the case for EOSDIS, so DCOM has not been investigated in great technical detail thus far, but it has not been excluded as an option, inasmuch as DCOM is perhaps the most current, best-supported implementation of DCE RPC today.

It may be that a limited UNIX port of DCOM could support a legacy DCE application. There may also be cases where a legacy DCE application could be ported reasonably easily to MS Windows, and then to DCOM. Such cases are exceptional, but deserving of consideration.

1.4.4 CORBA

Legacy applications that exist within heterogeneous computing environments, such as NASA's EOSDIS project, are not ideal candidates for migration to DCOM for reasons stated above. CORBA, as an architecture that has been designed from the beginning for distributed computing in heterogeneous environments, appears to be a more suitable migration target.

Progress has already been made in the area of DCE to CORBA migration. A commercial one-way DCE-CORBA bridge [6.1] has been available for some time now, and source-level tools were developed for automating a specific DCE-to-CORBA migration in 1997 [4.1]. Unfortunately, no in-depth details of these proprietary efforts exist in the public domain. Only the bridge is commercially available.

1.4.4.1 Limitations to Interoperability

From quite early in the process, some (but by no means all) of the leading vendors in OMG planned to support CORBA's standard interfaces using DCE. [2.6]

CORBA is partly based on DCE. This is mostly because of the substantial influence held by DCE vendors in OMG up to the finalization of CORBA v2.

IIOP's *common data representation* (CDR) is based on a subset of DCE's *network data representation* (NDR). The DCE Common Inter-ORB Protocol (DCE-CIOP) is another part of CORBA v2 that shows DCE's influence on CORBA. These apparent influences, however, actually represent compromises made to compensate for a decided move away from DCE in the core of CORBA v2. CDR may have been based on a part of NDR, but this did little for interoperability.

Hewlett-Packard and Digital Equipment Corporation used DCE RPC as the foundation of their CORBA implementations *OrbPlus* and *ObjectBroker*. These products were discontinued, and so was the participation of HP and DEC in CORBA product development. The DCE Common Inter-ORB Protocol (DCE-CIOP) remains within the CORBA specification [5.3], but it is not maintained, and is growing less and less compatible with the rest of CORBA, unused and virtually forgotten.

These events make migration of DCE systems to CORBA more challenging than it might have otherwise been.

CORBA and DCE have little in common at the service level. Likewise, a number of DCE RPC features were never addressed by CORBA. On the upside, the CORBA security service is somewhat compatible with that of DCE, and has recently been implemented by several ORB vendors and other developers.

The less a DCE application depends on DCE-specific features and services, the more suitable it is likely to be for migration to, or enhancement with, CORBA. Conversely, a DCE application that makes extensive use of DCE-specific IDL features and services is going to be difficult to port. Some of these problematic items are pipes, varying arrays, and DCE's distributed file system (DFS), distributed time service (DTS), and cell directory service (CDS).

In the NASA-EOSDIS case, the legacy application of interest makes use of only a small subset of these DCE-specific features and services. This appears to have also been the case for Schmidt, Gokhale, and Moyer [4.2].

1.4.4.2 Where value is added

Where migration to CORBA appears to be feasible, CORBA offers the legacy application a broad array of new features and services unavailable from DCE.

CORBA is more thoroughly object-oriented than DCE. Object-oriented extensions to DCE have been developed, such as OODCE, DCE++, dceObjects, and Objtran, but none of these is currently maintained or supported.

Several mature CORBA implementations exist on the market. They are not only supported, but maintained, and improving. As explained above, this is not the case for OSF DCE, and the only attempt to bridge OSF DCE to the Web (HTTP) has been abandoned. In contrast, CORBA's standard inter-ORB protocol, IIOP, is designed specifically for compatibility with HTTP.

CORBA supports bindings to languages other than C, for instance Java.

Though CORBA is challenged by DCOM, and RMI to a lesser extent, CORBA remains the leading distributed computing platform for heterogeneous computing. DCE applications that are not tightly tied to DCE-specific features and services may be

well positioned to make the move to CORBA, whether to CORBA as a whole or only to IIOP itself. This thesis discusses this issue in detail.

1.4.5 SOAP

The Simple Object Access Protocol (SOAP) is emerging as a standard Remote Procedure Call (RPC) format for object method invocation over the World Wide Web (specifically, over the HTTP protocol).

A primary reason for the growing support for SOAP is that it is being developed through the World Wide Web Consortium (W3C) as a means to using the Extensible Markup Language (XML) to invoke services over the Internet. SOAP RPC is not the only form of XML-based RPC. An earlier, openly available XML-based RPC, named "XML-RPC", was developed and is supported by one of the contributors to the SOAP specification⁷. These two protocols are much more alike than they are different.

SOAP benefits from its intrinsically close relationship to HTTP in more ways than one.

HTTP is the dominant high-level transport protocol on the Internet today. Even file and message transfer are often supported by means of HTTP rather than the traditional protocols used for these purposes (FTP and SMTP, respectively).

SOAP is designed with HTTP in mind, but does not depend upon HTTP. The SOAP specification includes a detailed discussion of *Using SOAP in HTTP*⁷. Because SOAP is designed with HTTP in mind, applications using SOAP to facilitate RPC don't need to modify their RPC calls to get them through most network firewalls.

The relationship between SOAP and the Web is more than just support from the same community. Since its beginning, HTTP has been closely tied with the popular document formatting and linking language HTML. Both were developed concurrently by the W3C. HTML's weakness as a non-extensible language has become readily apparent, so XML is being developed as a replacement, at least for the many situations where an extensible language is of use. SOAP, in turn, is one way that an extensible language such as XML can be used to package messages and procedure calls.

SOAP is not a standard, nor is it even a final specification; yet it is a very promising approach to messaging and RPC over the Internet. Its maturity may be too far off for immediate concerns, but on the other hand, early adopters of this technology stand to gain from the simple sophistication and adaptability of SOAP. This promising outlook requires that SOAP be investigated in depth herein.

⁷ David Winer of Microsoft and UserLand Software.

⁸ Section 6 of the specification

1.4.6 Summary

This thesis examines CORBA/IOP and SOAP in the most depth, because these technologies best satisfy the general requirements of a legacy, UNIX-based, DCE application such as EOSDIS. CORBA/IOP is much more mature than SOAP, but SOAP may become more interesting in the near future, especially for an Internet gateway strategy. DCOM should be watched closely as an alternative to CORBA, but the preconditions for this investigation do not support an in-depth investigation of DCOM at this time.

1.5 Migration and Integration Overview

The following diagram illustrates the full assortment of alternatives and extensions.

Figure 1: Migration Targets to be Addressed

At the center of the diagram is the DCE RPC protocol, which provides direct support for OODCE and DCOM clients and servers.

OODCE clients and servers can operate over CORBA by way of a wrapper layer.

CORBA clients and servers can be added to the application once OODCE servers are available on the ORB. They need not wait for OODCE servers to be recompiled with CORBA wrappers if a CORBA/DCE bridge is added to the application.

Likewise, OODCE clients and servers can operate over HTTP by way of a SOAP wrapper layer, and SOAP clients and servers can be added to the application once OODCE servers are available by way of SOAP/HTTP.

This diagram does not include other alternatives, such as pre-existing Internet gateways for OODCE, CORBA, and DCOM.

If DCE-RPC is removed from the core of a legacy OODCE application that has been ported over CORBA and SOAP, the architecture then resembles the following diagram.

Figure 2: OODCE programs running over CORBA and SOAP

New CORBA clients and servers are shown as an optional extension.

2 Principles of Migration

2.1 Levels at which Migration Occurs

Migration can be applied at different stages to distributed application development and execution, usually in combination. Each of these levels are addressed in this thesis:

1. Interface – an application's interfaces must often be reprocessed to generate modified stubs.
2. Source (Implementation in the server case) - source code must sometimes be modified if an application is to be migrated completely. Such modifications must be minimized.
3. Compile-time - Legacy application source code can be migrated in part by including headers that redefine terms for the purpose of migration.
4. Link-time - Libraries and other object code that need not be changed from one interface to another sometimes must be ported for the purposes of migration.
5. Run-time (bridging) - Procedure calls and returns at can be converted at execution time.

2.2 Adapters and Bridges

Porting interface implementation code from one middleware platform to another requires that data be translated to and from the target middleware platform's specifications. This translation is concealed in the stubs (or skeletons) generated for that interface implementation, or if not in the stubs, it must be present on the network. This thesis refers to the former as *source porting* or *source-level interoperability* [4.1-2], while the latter is referred to as *bridging*.

The obvious advantage of bridging is that legacy programs do not typically need to be recompiled to become accessible to clients using other middleware technologies. However, bridging does not provide a means to migrate code permanently such that the code would no longer be dependent on the bridge. Another disadvantage to bridging is that it can be slow, especially in the case of a dynamic bridge, which needs to interpret interfaces at run time.

The fundamental difference between source porting and bridging is the location of the translator. The translator can be compiled and linked into client and server binaries, or placed on the network. Whereas bridges may be dynamic or static, source-porting adapters are best kept static to optimize performance. Little is gained by statically linking dynamic adapters to legacy code. Dynamic linking of dynamic

adapters may be feasible as an alternative to a network-based bridge, but it seems more likely that such a job is better left to a dedicated network resource.

Bridges may be implemented at the interface level or at the protocol level. In other words, a bridge may be implemented as a proxy application or at a lower level, as a protocol converter (for example, IIOP-DCE/RPC).

It may be that the two approaches can be combined in a complimentary fashion; using a bridge for basic interoperability, and source porting to achieve permanent, bridge-independent interoperability. For example, the *Inprise DCE-CORBA Bridge* can be accompanied by the source porting strategy described herein, or a bridge can be devised based on the same source porting adapters.

To investigate the feasibility of developing such a bridge from source porting adapters, a SOAP-DCE bridge has been developed for this investigation, and will be described herein.

2.3 Achieving Source-level Interoperability

Any client or server using RPC middleware is going to exhibit the following structure: application code makes or receives a remote procedure call. The call parameters and return value are marshaled and un-marshaled by stub (or skeleton) code just below the application code layer. Beneath this data translation layer is a broker and transport layer. By *broker* is meant the facility provided by middleware that directs commands to their intended targets.

In the case of DCE, stubs are generated for each application service interface. Every DCE client program and server program is linked to such a stub. For source porting to be achieved, these stubs can be replaced by look-alike stubs that appear to be DCE stubs to the application programmer yet actually utilize another middleware technology, such as CORBA or SOAP.

2.4 Limitations to Interoperability

The proposed strategy, whether it be applied to a migration to IIOP or SOAP, presumes the source legacy application does not use specific features that either do not map easily to the target middleware or are associated with services that are supplementary to, or beyond the scope of, the RPC model.

2.4.1 Special DCE RPC Features

2.4.1.1 Context Rundown Procedures

A DCE context handle may be orphaned if the client that had it created fails to free it or if the client connection fails. DCE provides a rundown procedure facility for such cases that is emulated by the migration architecture, but not fully replicated.

2.4.1.2 Pipes

CORBA and SOAP have no feature that corresponds to DCE pipes, nor does the translation mechanism proposed by this thesis support DCE pipes. This does not mean that pipe support cannot be added, only that it (1) would involve significant work and (2) was not needed by the legacy application at hand. If pipe support were necessary, it

could probably be provided by means of OMG Audio/Video Streaming, though this OMG specification is not part of CORBA, nor is it implemented in the major commercial ORBs.

2.4.1.3 Idempotent Functions

An idempotent function is called by DCE as many times as it takes DCE to receive a response from the function implementation. This is useful when DCE is using UDP, a datagram protocol that doesn't guarantee message delivery as TCP does. Because the unreliability of UDP is not an issue for CORBA or SOAP (over HTTP), lack of idempotent support is not a problem.

2.4.1.4 Pointers

CORBA does not support the passing of pointers to or from methods. This does not mean that pointers cannot be passed, but only CORBA does not specify how to do so in a portable manner. This limitation is not seen to be critical, because encoding and decoding of pointers can be controlled at both ends of a transmission. If the objective of this thesis were to develop a specification or to propose a standard, a portable, general purpose translation scheme would be required. The CORBA data type that best corresponds to a pointer is an octet sequence. CORBA valuetypes can also be used to

pass functional pointers (that actually point to data), but that is not necessary. Where objects must be passed by value, the conversion framework does so without any dependence on CORBA valuetypes.

The issue is the same for SOAP.

2.4.1.5 ACF Attributes

ACF attributes depend on special DCE features that complicate the task of migration. For example, the `comm_status` and `fault_status` attributes can be used by a DCE client to obtain error and exception information without formal exception handling code. This can be supported by a migration layer such as the one presented herein, but it means more work. ACF attributes are not supported by the converter developed in support of this thesis.

2.4.1.6 Varying Arrays

The CORBA specification does not support varying (partially transmitted) arrays, so DCE/CORBA adapters generated for a specific interface must pass additional information across an ORB to support varying arrays. The SOAP specification *does* support varying arrays.

2.4.1.7 Conformant Types

Conformant arrays and structures are sized at runtime. They complicate the IDL parsing process by requiring that the IDL parser check for IDL attributes and then associate parameters or struct members through those attributes that would otherwise be non-related. Conformant structures complicate matters further by adjusting the size of an array that is declared as a single-element array in C. This means that the struct itself is resized to accommodate one of its members. This makes the compilation process even more involved. The conversion utility developed in support of this thesis provides support for these types that is sufficient for the purpose of the thesis.

2.4.2 Supplementary Services

2.4.2.1 Security

DCE provides a security service that most legacy DCE applications must be expected to utilize, though the particular NASA application that motivated this work does not depend on the DCE security service. The security services of DCE and CORBA have enough in common that a security service mapping could be included in

the migration strategy proposed by this thesis, however mapping the DCE security service to CORBAsec was not considered critical in this case.

Security for SOAP would be provided by way of HTTPS/SSL. This may or may not be feasible, depending on the particular DCE application. Given the lack of industry support for CORBA Security at this time, this may be seen as an advantage for SOAP.

2.4.2.2 Threads

Neither CORBA nor SOAP provides any thread library that corresponds to DCE's portable thread facility. Therefore a thread library that resembles DCE threads must be developed. This was a simple task on the selected target platform (Windows), thanks to the standardization of thread interfaces.

2.4.2.3 Distributed Timing

DCE provides a distributed timing service (DTS), but it is not tied into DCE RPC. It is easily replaced by other existing COTS or in-house products.

2.4.3 Non-RPC Services

DCE provides a distributed file system (DFS) that some legacy applications may depend upon, though this was not an issue in the case at hand. By virtue of the fact that DFS is a non-RPC technology, it follows that any possible dependence on DFS to be a platform dependence issue rather than a middleware issue.

3 Migrating to Sun/ONC RPC

3.1 Introduction: Feature Support

This analysis of DCE to ONC migration begins with a feature-by-feature introduction to both environments.

The following feature tables list some of the feature-set differences between the two RPC technologies. Though the two use the same basic development paradigm, and have moved closer over the years, it is clear that they are far from compatible. The two use different services, different parameter rules, different data representations, different data transfer rules, different message framing techniques⁹, and as will be detailed next, different programming interfaces.

⁹ Power Programming with RPC, section 2.7.3.

Table 1: OSF and ONC Data Representations

	OSF/DCE	Sun-ONC
MACHINE-INDEPENDENT DATA REPRESENTATION		
En-Route Data Representation	Network Data Representation (NDR)	eXternal Data Representation (XDR)
Byte ordering	Receiver-makes-it-right	Intermediary data format; favors big-endian systems
Data structure translation between compatible machines	Run-time	Compile-time?
Parameters	input, output and input/output	Supports one input parameter and one output parameter per procedure (structs allowed).
Pipe, array, pointer, and context handle data types¹⁰	Yes	No
Pointer-chasing support (through structures)	No	Yes

¹⁰ Power Programming with RPC, section 2.7.5.

Table 2: OSF and ONC RPC Mechanisms

RPC Mechanism		
Transport Independence	No	Yes (optional, via the SVR4 TLI)
Low-level network protocols	TCP, UDP ¹¹	TCP, UDP, OSI, SPX/IPX
Callback support in protocol compiler	Yes	No
Stateful behavior	Stateful	State of transport
Threads ¹²	DCE Thread Library (based on DEC CMA?; supports POSIX)	Sun LWP, HP CPS, DEC CMA
Lightweight Threads	Yes (where available)	No
Time-out and retry control	No	Yes

Table 3: OSF and ONC Protocol Compilers

Protocol Compilers	IDL ¹³	RPCGEN.NEW and RPCTOOL
Protocol/Interface Definition Language	IDL	RPCL (based on XDR)
Stub Integrity	Client and server stubs may need modification, depending on authentication.	Server stubs may need modification, depending on authentication.

¹¹ Guide to Writing DCE Applications, pages 59-60. TCP replaced Domain DDS in DCE. See Power Programming with RPC, page 27.

¹² Power Programming with RPC. Pages 28-29.

¹³ Based on NIDL, the NDS protocol compiler.

Table 4: OSF and ONC Network Services

NETWORK RESOURCE NAMING SERVICE	CDS and GDS (X.500 implementation)	NIS
NETWORK-WIDE BINDING SERVICE	Yes	No ¹⁴
BINDING TYPES	Automatic, implicit, and explicit. ¹⁵	Low-level and high-level functions.

¹⁴ Power Programming with RPC. Sections 2.7.5 and 2.7.7.

¹⁵ DCE's explicit binding is used by OODCE (though hidden in the OODCE stubs). It is also considered necessary when migrating to CORBA.

3.2 Interface Compatibility

3.2.1 Interface Definitions

Interface migration can be difficult when migrating from DCE to ONC RPC, because the languages RPCL and IDL have significant differences.

3.2.2 Application Programming Interfaces

ONC RPC provides both high-level and low-level programming approaches, however, because the high-level approach limits the program to the UDP protocol, doesn't provide any security facilities, and so on, only the low-level approach will be considered herein.

3.2.2.1 OSF/DCE

The following tables list some of the more important functions used in DCE development. Note the simple requirements for the DCE client, and the comparatively complex requirements for DCE servers. Also note the numerous references to binding in the function names.

Table 5: OSF RPC Client Library Calls

Required Client Library Calls	Optional Client Library Calls
	do_import_binding (explicit binding)

Table 6: OSF RPC Server Library Calls

Required Server Library Calls	Optional Server Library Calls
rpc_server_register_if	CHECK_STATUS
rpc_server_use_all_protseqs (or similar)	
rpc_server_inq_bindings	
rpc_ns_binding_export	
rpc_ep_register	
rpc_binding_vector_free	
rpc_server_listen	

Table 7: OSF RPC Binding Calls

Interpreting Binding Info (implicit and	Importing a Binding Handle
--	-----------------------------------

explicit binding)	
rpc_binding_to_string_binding	rpc_ns_binding_import_begin
rpc_string_binding_parse	rpc_ns_binding_import_next
rpc_string_free	do_interpret_binding
	rpc_binding_free
	rpc_ns_binding_import_done

Table 8: OSF RPC Binding Calls (cont.)

Setting a Binding Handle from Strings
rpc_network_inq_protseqs
rpc_string_binding_compose
rpc_binding_from_string_binding
rpc_string_free
rpc_protseq_vector_free

3.2.2.2 Sun-ONC

The following RPC library functions are used by clients in accessing ONC RPC servers.

Table 9: ONC RPC Client Library Calls

Required Client Library Calls	Optional Client Library Calls
clnt_create	clnt_control
clnt_call ¹⁶	clnt_pcreateerror
	clnt_perror / clnt_perormo
	perror
	clnt_destroy

The svcxxx_create library functions are needed if a server is to acquire its transport handle (for instance, if specifying the TCP transport).

¹⁶ The form this takes is unclear. The description on page 80 of Power Programming with RPC and the example code and description on pages 86-87 appear to contradict.

Table 10: ONC Server Library Calls

Optional Server Library Calls
svctcp_create
svcudp_create
svc_register
xdr_free

Upon comparison of the two application programming interfaces, it is easy to see that the detail of programming against the interfaces differs greatly. A further review of the use of these interfaces only serves to underscore the difference. Furthermore, these well-documented public APIs are only a small fraction of the differences between DCE and ONC functions, if one considers the mass of functions utilized in the client and server stubs.

3.3 OODCE-over-ONC

OODCE-like object wrappers could be re-implemented over Sun RPC.

This may be more feasible than migrating directly from DCE RPC to Sun RPC.

In abstract terms, OODCE would be a better match for Sun RPC because it can be used as a wrapper for Sun RPC.

In more practical terms, this approach would not abrogate the need to reimplement the DCE RPC API over Sun RPC, because legacy DCE applications such as ECS directly access the DCE RPC API. This would also be an issue for CORBA and SOAP migrations.

The difference with an OODCE-over-ONC wrapper strategy is that, though the labor investment would be similar, the potential for added value would be relatively small.

3.4 Conclusions

It may be that a viable DCE-to-ONC migration scheme can be engineered for a reasonably inclusive subset of legacy DCE applications. Whether the same can be done for legacy OODCE applications is less likely, considering that OODCE itself is highly dependent on various aspects of DCE RPC. On the other hand, it may be that OODCE-

like object wrappers could be designed for RPC. In any case, the translation between the two RPC interfaces and mechanisms would not be simple.

The most relevant question is this: what does an application gain by migrating from DCE to RPC? Simplicity is the chief comparative virtue of ONC RPC, but simplicity is not what one would gain from such a complex migration. One must have something positive to gain from ONC. The fact that ONC RPC is freely available is attractive, but feature-wise, what is there to gain? Not interoperability, as long as DCE offers the advantage of DCOM interoperability. In addition, ONC RPC lacks many DCE features, and offers no equivalent to OODCE.

4 Migrating to DCOM

4.1 General Discussion

"The DCOM network protocol is often called Object RPC or ORPC to emphasize the close relationship between RPC and the DCOM protocol at the network level. ORPC highly leverages the functionality of the OSF DCE RPC network protocol. For example, the authentication, authorization, and message integrity/encryption features of RPC are present in ORPC." [8.3]

DCOM, as middleware, is perhaps the best fitting target for migration of DCE applications, because it is based on DCE RPC and it is actively supported and maintained by Microsoft. With DCE RPC operating beneath both DCE and DCOM, they are effectively joined by the best kind of "Inter-ORB Protocol", because DCE RPC happens to be the internal RPC protocol of each.

It will be demonstrated below that the advantages of porting to DCOM extend beyond runtime; the DCOM development environment is quite friendly to DCE programming.

4.2 Writing DCE Programs for DCOM

4.2.1 Analysis

DCOM uses the Microsoft Interface Definition Language (MIDL) to specify interfaces. MIDL is an extension of OSF IDL, and the DCOM MIDL compiler includes an /OSF switch for processing OSF IDL files [8.4].

DCE support in the DCOM development environment does not end at MIDL. The Microsoft RPC API closely resembles the DCE RPC API. The following table enumerates the Microsoft RPC support for some of the DCE RPC function classes.

Table 11: DCOM Support of the DCE RPC API

Function Prefix	MS RPC¹⁷	Common [1.2]	Total [2.2][2.7]
dce_error_	(all)	1	1
dce_cf_	0	1	11
rpc_binding_	(all)	7	13
rpc_ep_	(all)	3	4
rpc_mgmt_	(all)	7	16
rpc_network_	(all)	1	2
rpc_ns_	(all)	15	39
rpc_object_	(all)	0	3
rpc_server_	(all)	9	11
rpc_sm_	(all)	0	10
rpc_ss_	10	2	11
rpc_string_	(all)	3	3
uuid_	(all)	5	8

In all, there are a large number of DCE commands that are not supported by Microsoft RPC, but the great majority of common RPC commands are supported.

Microsoft RPC uses a different naming convention for functions than that used by DCE RPC. Microsoft uses a leading uppercase convention, wherein each name component begins with an uppercase character. DCE RPC uses an all-lowercase, underscore-separated convention, wherein the components of a name are separated by the underscore character.

¹⁷ http://msdn.microsoft.com/library/psdk/rpc/pr-func1_6v03.htm

Data types are given the same names in DCE RPC and Microsoft RPC.

Constants are also named with the same convention, except that Microsoft RPC constants are entirely uppercase and DCE RPC constants are entirely lowercase.

DCE RPC library functions return status in a parameter, whereas Microsoft RPC library functions return status as the function return value.

These rules are applied consistently for both DCE RPC and Microsoft RPC, so conversion between the two is a simple task.

4.2.2 Porting a Simple DCE Client/Server Pair to DCOM

To support this thesis, a simple DCE client/server pair, based on the NASA EOSDIS Core System (ECS) subscription server interface, was ported to DCOM. The ECS IDL files were processed by the MIDL command, using the /OSF switch. All the code changes specified in the previous section were applied to the source code. In this case, the changes were applied manually, though a source conversion tool would have been easy to develop.

Finally, the sources were compiled against MS-RPC headers (`rpc.h`, `rpcndr.h`, and `rpcdce.h`) and linked against MS-RPC libraries (`rpcrt4.lib` and `rpcns4.lib`). The code was linked to `rpcns4.lib` because it makes name service calls.

The client and server programs share access to the following files:

<code><interface>.h</code>	Header output by MIDL for <code><interface>.idl</code>
<code><types>.h</code>	Header output by MIDL for <code><types>.idl</code>
<code>check_status.h*</code>	A standard DCE status reporting macro
<code>shared.c/h</code>	Common utility code/header

The server consists of the following files:

<code><interface>_s.c</code>	Server stub output by MIDL for <code><interface>.idl</code>
<code><interface>impl.c*</code>	Server implementation code called by stub
<code>server.cpp*</code>	Main program entry point, consisting of server object management code
<code>Subscription.c/h</code>	Example Subscription code/header; not dependent on DCE

The client consists of the following files:

<code><interface>_c.c</code>	Client stub output by MIDL for <code><interface>.idl</code>
<code>client.cpp*</code>	Main program entry point that finds the server and calls the client stub
<code>do_import_binding.c*</code>	Standard DCE binding import function
<code>do_interpret_binding.c*</code>	Standard DCE binding interpretation function

*See code listings under "DCE over DCOM".

4.3 OODCE

The above analysis and examples apply to conventional, procedural DCE applications.

Porting OODCE applications requires more work, but it is a much simpler task to port an OODCE application to DCOM than it is to port the same application to CORBA or SOAP. There are three primary reasons for this:

- OODCE and DCOM use the same data types.
- OODCE and DCOM support the same RPC features.
- OODCE stubs are built upon DCE stubs, which can be generated by MIDL.
- OODCE library facilities such as DCEServer and DCEPThread are built upon DCE facilities.
- OODCE applications typically make DCE RPC API calls, most of which are supported by DCOM.

Data type conversion is perhaps the single most difficult aspect of porting OODCE applications, depending on how important it is for a ported application to use the full selection of RPC features. If a legacy application makes heavy use of varying arrays, pipes, context handles, ACF attributes, and so forth, then the second factor (RPC feature support) is the more critical factor, and the advantage of porting to DCOM is all the more pronounced.

4.4 Moving to DCOM Interfaces

It has been demonstrated above that DCE and OODCE applications are relatively simple to port to the DCOM runtime environment. Some applications may be further served by porting one or more of their interfaces to DCOM, to take advantage of features specific to DCOM.

The necessary modifications are:

1. The `oaidl.idl` and `ocidl.idl` files must be imported.
2. The MIDL header attribute `object` must be added to the IDL header.
3. Any data types used as distributed function parameters cannot be defined as handles or derivatives of the `void` type.
4. Binding handles are not used, and they must be removed from the signatures of distributed functions.
5. The `idempotent` attribute is not supported in a compatible manner and must be removed.
6. Rather than allowing distributed functions to return a variety of data types, DCOM requires that distributed functions return a value of the 32-bit `HRESULT` type.
 - a. This means that DCE functions that return data must typically be redesigned to pass that same data back by way of an `[out, retval]` parameter.
 - b. The `[out, retval]` parameter must be the last parameter.

To illustrate, the first lines of the DCE IDL for the ECS Subscription interface are shown below. Elements to be removed for DCOM compatibility are struck through.

```
[
  uuid(006a1076-0742-1147-baa6-9b9d6514aa77) ,
  pointer_default(ptr) ,
  version(1.0)
```

```

]

interface DsSdDCESubscrIDL
{
import "DsShDCETypes.idl";

typedef {context_handle} void* DsSdDCESubscrHandle;

DsSdDCESubscrHandle CreateNewSubscription(
    {in} handle_t h,
    [in, size_is(len)] byte data[],
    [in] long len );

```

After the IDL is converted to DCOM, it appears as follows (additions in bold):

```

import "oaidl.idl";
import "ocidl.idl";

[
    object,
    uuid(006a1076-0742-1147-baa6-9b9d6514aa77) ,
    pointer_default(ptr) ,
    version(1.0)
]

interface DsSdDCESubscrIDL : IUnknown
{
import "DsShDCETypes.idl";

typedef long DsSdDCESubscrHandle;

HRESULT CreateNewSubscription(
    [in, size_is(len)] byte data[],
    [in] long len,
    [out, retval] DsSdDCESubscrHandle* subscr );

```

Some of these changes alter function prototypes in the <interface>.h file significantly. A function no longer returns the data that results from the operation it performs, but rather returns a status code. The data output by the operation is returned through an out parameter. This parameter is a pointer to the data type that was

previously returned (by the DCE interface), unless the DCE data type was a handle type, in which case the handle type would be replaced by an unsigned long integer, or something similar.

DCE Function Signature:

```
DsSdDCESubscrHandle
CreateNewSubscription(
    handle_t h,
    byte __RPC_FAR data[ ],
    long len );
```

DCOM Function Signature (for C):

```
HRESULT ( STDMETHODCALLTYPE __RPC_FAR
*CreateNewSubscription )(
    DsSdDCESubscrIDL __RPC_FAR * This,
    byte __RPC_FAR data[ ],
    long len,
    DsSdDCESubscrHandle __RPC_FAR *subscr );
```

DCOM Function Signature (for C++):

```
HRESULT STDMETHODCALLTYPE
CreateNewSubscription(
    byte __RPC_FAR data[ ],
    long len,
    DsSdDCESubscrHandle __RPC_FAR *subscr);
```

In the code comparison above, observe the replacement of the binding handle (handle_t) with an object pointer (<interface>*) in the C language version of the DCOM function signature, and the absence of such object handles in the C++ version of the DCOM function signature.

Developing an automated process for moving IDL and client and server code from DCE to DCOM would not be trivial, for the reasons stated above, but it would not be as difficult as porting to CORBA or SOAP, for the reason that two of the primary advantages previously cited for porting DCE applications to the DCOM runtime also apply to converting DCE applications to DCOM applications.

1. DCE and DCOM use the same data types.
2. For the most part, DCE and DCOM support the same RPC features.

4.5 Added Value

DCOM provides added value to DCE applications in several aspects.

DCOM provides object orientation and an unsurpassed component model. Though COM's object model is not based on the same object paradigm as CORBA, it is perhaps the more useful, because DCOM offers the foremost component model on the market, and brings that component model to the desktop in a way that is easier to develop and more powerful than anything supported by any competing middleware.

DCOM development is strengthened by Microsoft's development tools, perhaps the most capable development tools on the market.

DCOM is supplemented by a continually expanding selection of services and features, whereas DCE's selection of features and services froze years ago.

Though the Windows NT is not highly regarded for reliability, it is improving, and Windows is a very successful client platform. The client side of the client/server equation should not be underestimated. The dominance of Windows in the client market means that, though DCOM may not be fully ported to UNIX on the server side, it is effectively ported to the great majority of client systems.

4.6 Challenges

4.6.1 The Risk of Vendor Lock-In.

DCOM's primary drawback is that it is a Windows-specific technology. This gives rise to a significant portability issue, and also a problem for mission-critical computing, as Windows NT is not yet regarded to be as robust as some UNIX and mainframe systems with respect to reliability and fault tolerance.

The portability issue is not a straightforward one, however. A useful measurement of portability begins with the specific needs of a particular application. For instance, when considering migration of a UNIX-based legacy system to DCOM, one must consider the fact that, though DCOM has been ported to a degree on several UNIX platforms, there's really no plan for DCOM to be thoroughly ported to any non-Windows platform. Though basic DCOM facilities have been ported, several critical system facilities that mature DCOM applications depend upon will not be ported.

On the other hand, the mere fact that a complete port of DCOM is not on Microsoft's agenda should not stop anyone from considering migration to DCOM. Legacy DCE applications don't depend on those Windows services not being ported to UNIX. Though a port may be incomplete from the perspective of a Windows application with

all the expected Windows entanglements, the port may well be considered complete in practical terms from the perspective of a legacy DCE application.

Furthermore, an incremental migration from DCE to DCOM may not depend greatly on the porting of DCOM to UNIX. The reason for this is that DCOM is highly compatible with DCE over the wire, regardless of what local operating systems DCOM or DCE might be operating upon. The essential characteristic of distributed computing is that of applications being distributed across a network of machines. Since DCOM and DCE speak the same network language, what better middleware is there to integrate with DCE than DCOM?

So it is that, in terms of real-world projects, the middleware that has proven most feasible as a migration target for legacy DCE systems has proven to be DCOM.

Though it may also be true that choosing DCOM causes the application to become dangerously dependent on the Windows platform and Microsoft, yet in another respect an incremental migration to DCOM diversifies an application. The migration may require training, the hiring of Windows developers, and so on, yet many organizations may find themselves invested in Windows already at the desktop. The situation becomes more hazardous as larger portions of the application become Windows-dependent.

4.6.2 Platform Support

4.6.2.1 Ports of DCOM to Non-Windows Platforms

➤ Ports by Compaq

- Tru64 UNIX¹⁸
- OpenVMS

➤ Ports by Software AG¹⁹

- Solaris 2.7
- AIX 4.2, 4.3
- HPUX 11.0
- Tru64 UNIX 4.0
- OS/390

4.6.2.2 Services not ported to UNIX

- Microsoft Transaction Server (MTS)²⁰
- Microsoft Message Queuing (MSMQ) Service
- Windows Registry
- Windows Help

¹⁸ Formerly known as Digital UNIX and DEC OSF/1.

¹⁹ NASA's ECS is reportedly based on an SGI/IRIX implementation of HP's OODCE. Microsoft and Software AG have no plans for supporting DCOM on SGI/IRIX. No current commercial CORBA ORBs with C++ bindings support IRIX.

²⁰ See COM and CORBA, Side by Side – Architectures, Strategies, and Implementations; by Jason Pritchard, Ph.D.; page 23.

4.6.3 Measuring the Impact of Missing Windows Services

The Windows Registry and Windows Help have not been ported to UNIX, but this fact is of no consequence to a legacy DCE application being ported to DCOM within the UNIX environment.

Of course many legacy DCE applications would be able to make use of advanced services such as MTS and MSMQ on UNIX, but it cannot be said that they would depend on these specific Microsoft services.

4.6.4 Recent Enhancements to DCOM Portability

Windows component security has traditionally been incompatible with the UNIX (Kerberos) security model. This has meant that components should not be made public, but rather hidden behind a Windows server. This was resolved, however, with the introduction of Kerberos with Windows 2000.²¹

4.6.5 A Long View

The portability problem would arise more seriously later if any attempt were made to port any part or all of an application to a non-Windows platform. This is because Microsoft development tools make it very difficult to develop portable software. Even with non-Microsoft tools, such as Inprise Delphi, the DCOM development process itself is non-portable by nature.

This one-way portability has not come about by circumstance, but by design. Microsoft has great interest in attracting migration to DCOM, but Microsoft has no interest in supporting migrations away from Windows NT.

²¹ See http://msdn.microsoft.com/library/techart/msdn_unixcom.htm#unixcom_questions.

"The trade-offs to using DCOM and MTS are familiar ones: You gain a solid, consistent environment, mature development tools, and a distributed component model supported by a huge installed base. On the downside, you lock yourself into a relationship with a single vendor, Microsoft. Despite the company's recent support for "COM on Unix," you can rest assured that support for Unix will always be minimal to induce companies to migrate to Windows-based servers. Microsoft representatives have repeatedly stated that they have no interest in making Unix a stronger platform..." [5.5]

To be fair, it must be mentioned that vendor lock-in is not a problem that is exclusive to Microsoft products. It has been a significant problem with CORBA implementations as well, though technologies such as IIOP and POA have improved CORBA application portability and ORB interoperability.

Java application development is not immune from the lock-in issue either so long as Sun Microsystems continues to maintain firm control of Java technologies.

4.6.6 Java Interoperability

Another portability problem for DCOM exists in its Java interface. Java programs that utilize DCOM must do so within Microsoft's JVM, which is completely unsupported on non-Windows platforms. Hence, though Java is designed for portability, any Java code that uses DCOM could not be ported to UNIX at a later date.

It may be that DCOM will never inter-operate with non-Microsoft Java environments, but since Java is leaning increasingly toward the server side, this may never become an issue, except with back-end servers layered behind Java middleware. That far back-end will probably be the last to see migration to DCOM, if it ever does.

4.6.7 Scalability

Though DCOM has been proven to be scalable enough to suit large enterprise operations, DCOM may introduce scalability issues at the Internet scale. This is because DCOM servers, by default, automatically ping their clients at two-minute intervals to verify connections [9.13]. This behavior may become costly when client/server connections become very numerous, as is the case for the Internet.

4.6.8 The Client

Legacy applications are often weakened by their primitive client interfaces. NASA's EOSDIS is no exception. Without powerful, easy-to-use front ends, the most powerful of services may never get close to their potential usefulness. There is little doubt that DCOM is in a position to enhance the client tier of EOSDIS dramatically, without exposing the core system, ECS, to significant risk of vendor lock-in. Many organizations with large investments in mainframe and UNIX back-ends are adopting Windows as their client platform of choice. The strength of Windows as a client is enhanced by the fact that Windows can contribute to middle tiers as well, where application logic can be rapidly implemented. Windows may not yet be robust enough for mission-critical services, but the extent to which this is true has been diminishing rapidly in recent years.

Microsoft Windows is well equipped to be a DCE client platform, but this does not mean that DCOM is equally well positioned as a DCE migration target. For a Windows DCE client to become a DCOM client, it needs more than a DCE-based protocol; it needs a DCOM server, or at least a DCOM-to-DCE bridge. Such a bridge should be simple to develop. As explained above, the data types are the same, and the message format is the same, but DCE and DCOM method signatures are not alike, so some translation is necessary.

In the EOSDIS case, there is more to migration than enhancing the client side of the system with *Windows* clients. The need for migration is motivated by issues on the server side, so whatever advantages Windows clients have to offer are at best supplemental, but more often irrelevant.

4.7 Conclusions

In its core characteristics, DCOM is arguably the best migration candidate for legacy DCE applications. Unfortunately, the issue is complicated by platform support considerations.

The weakest link in any migration strategy is application source code modification, and any migration from a UNIX-based DCE application to a Windows-based DCOM application would require intensive source code modification work. The principal limitation to the value of having DCOM based on DCE is that one cannot expect a legacy DCE application to be written for *Windows*.

UNIX ports of DCOM exist, but they are partial, and not supported by Microsoft.

Since DCOM is not an open standard, the fact that Microsoft has complete control over its direction may put the long-term compatibility, and hence marketability, of these third-party UNIX ports at risk.

Some legacy applications are more portable to Windows than others, and some may be well positioned to move partly to Windows while using DCOM ports on UNIX while phasing out DCE. There are perhaps as many solutions as there are legacy applications.

What is the best technical choice depends largely on the application.

5 Migrating to CORBA

5.1 Challenges

The problem with migrating from OODCE to CORBA is that the two technologies are incompatible at all stages:

- **Pre-compilation**
 - DCE IDL and OMG/CORBA IDL are related, but incompatible.
- **Compile-time**
 - Interfaces to OODCE and CORBA stubs and services are incompatible, so application source code is incompatible.
 - Some OODCE applications may call DCE functions directly, requiring further re-coding or even switching to a DCE-to-CORBA strategy.
- **Link-time (in some cases link may occur at run-time)**
 - Libraries are incompatible. Though application source code differences may be bridged, libraries (services and stubs) must be replaced.
- **Run-time**
 - CORBA and OODCE applications use incompatible wire protocols (DCE/RPC and IIOP).

In cases where an OODCE-to-CORBA migration proves impractical due to dependency on direct references to DCE libraries, a DCE-to-CORBA strategy may be necessary. In this case the problems are:

- **Pre-compilation**
 - OSF/DCE IDL and OMG/CORBA IDL are related, but incompatible.
- **Compile-time**
 - Interfaces to DCE and CORBA stubs and services are incompatible, so application source code is incompatible.
- **Link-time (in some cases link may occur at run-time)**
 - Libraries are incompatible. Though application source code differences may be bridged, libraries (services and stubs) must be replaced.
- **Run-time**
 - CORBA and DCE applications use incompatible wire protocols (DCE/RPC and IIOP).

5.2 Solutions

5.2.1 Bridging

A DCE-to-CORBA bridge can be employed to solve part, and in simple cases all, of the problem. A bridge provides an immediate, somewhat seamless approach to integration of incompatible middleware platforms. Another important advantage to bridging is that it is the only DCE/ODDCE-to-CORBA migration strategy that provides commercial off-the-shelf (COTS) support.

5.2.1.1 DCE-CORBA Bridges

5.2.1.1.1 Inprise DCE-CORBA Bridge

This is currently the only DCE-to-CORBA bridge product available. Indeed, it's the only product that even resembles a DCE-to-CORBA bridge. It is essentially a one-way bridge that enables IIOP clients to use DCE/RPC services.

Inprise classifies their bridge to be a dynamic bridge, however it falls under the "on-demand" classification described by Yang and Vogel [6.5] of DTSC²². According to Yang and Vogel, a dynamic bridge is one that utilizes dynamic invocation interfaces, a technology that is not supported by DCE, yet appears to be possible. [6.6]

5.2.1.1.2 DTSC On-Demand DCE-CORBA Bridge

In 1996, a pre-release on-demand bridge was announced by DSTC. Because more recent information is not available, it is presumed that this project has been discontinued.

5.2.1.1.3 DCE Common Inter-ORB Protocol (CIOP)

This half-bridge sub-specification of CORBA was once supported by Hewlett-Packard's ORB Plus and Digital Equipment Corporation's ObjectBroker. Both of these ORB products have been discontinued.

DCE-CIOP appears to have been suffering from terminal neglect over the last several years since IIOP appeared. The OMG doesn't appear willing to remove DCE-CIOP

22 CRC for Distributed Systems Technology (DSTC), University of Queensland (Brisbane),

from the CORBA specification, yet they don't appear willing to keep the DCE-CIOP sub-specification compliant with changes in CORBA.

DCE-CIOP once represented the OMG's intention to use DCE/RPC as its primary inter-ORB protocol. Now, the advent of the Internet and IIOP has left DCE completely out of the CORBA vision. The demise of DCE-CIOP has left CORBA much less interoperable with DCE and DCOM. Perhaps this is simply due to lack of demand, or perhaps it is due to lack of cooperation between the OMG and Microsoft.

5.2.1.2 Problems with Bridging

Because the Inprise DCE-CORBA Bridge is the only commercially available bridging option, it will be the only such product discussed herein from this point on.

The usefulness of the Inprise DCE-CORBA Bridge is limited by several significant restrictions:

- The Inprise DCE-CORBA Bridge is a one-directional bridge, meaning that DCE components (whether legacy or newly developed) cannot access CORBA services.
- On-demand bridging imposes a performance hit on CORBA clients attempting to use DCE services. The performance hit is likely to be a one-time hit, as the mechanism behind on-demand bridging is similar to just-in-time compilation. On-demand bridging is not as flexible as true dynamic bridging, nor is it as susceptible to performance problems.
- Bridging provides no direct means of migrating DCE code to CORBA.
- As with any other migration strategy, the usefulness of on-demand bridging is limited by the legacy application's dependence on CORBA-incompatible aspects of DCE.

5.2.2 Alternatives to Bridging

Instead of relying on a bridge between CORBA and DCE domains, migration can be accomplished by modifying the legacy application to operate within the CORBA domain. This can be accomplished using several different strategies:

- Modify application code to operate with CORBA skeletons/stubs. This alternative, which would typically involve code conversion utilities, is usually too difficult to implement with legacy applications of any significant size or complexity.
- Re-implement the DCE IDL compiler or OODCE IDL++ compiler²³ to generate stubs that support IIOP rather than DCE/RPC. The advantage of this strategy would be that it wouldn't require the high-level compatibility layer of the preceding alternative. However, getting the DCE or OODCE stub interface to work for an IIOP implementation might require lower-level overhead. Implementing CORBA support at the IIOP level would probably mean developing a hybrid between *idl++* and *idl2cpp* from the ground up, not to mention developing a suite of support headers and libraries.
- Design a compatibility layer to be placed between legacy DCE or OODCE application code and CORBA skeletons/stubs. Implement this compatibility layer with adapter modules that are partially shared code and partially interface-specific. This is probably the simplest strategy among these alternatives. Much of the hard work is left to DCE/OODCE and CORBA IDL compilers. This is the strategy of choice for this thesis, and will be described in detail herein.
- Follow the same strategy as the previous item, but from a DCE-to-CORBA perspective. See the section entitled "Previous Research" for details.

²³ The source code for *idl++* is the property of NASA, and is not immediately available at this time.

- **Modify the OODCE *idl++* compiler so that it generates CORBA stubs and skeletons instead of DCE stubs, while not altering the interface it currently generates. This is an elegant solution, but it requires extensive modification of compiler code; effectively, a hybrid of the *idl++* and *idl2cpp* compilers. It is also an inflexible option, as it doesn't leverage diverse CORBA implementations, but rather it supports only its own.**
- **Develop a utility that generates CORBA stubs and skeletons that can be placed under the interfaces generated by *idl++*. This involves modifying the CORBA *idl2cpp* compiler rather than replacing the OODCE *idl++* compiler, and would probably be more difficult to accomplish. This option is inflexible for the same reason that the previous option is inflexible: it enforces a single CORBA implementation.**

5.2.3 Source-Level Interoperability via Adapters

The closest thing the EOSDIS Core System (ECS) provides to a universal API or abstraction layer is OODCE. This layer has two important strengths:

- OODCE provides a simple interface, relative to DCE.
- It is reported that very little EOSDIS application code undermines the OODCE interface by making direct calls to DCE.²⁴

The most natural approach to porting OODCE applications to CORBA is to encapsulate CORBA client stubs and server skeletons within OODCE-style classes. Such encapsulation would be performed by an *idl++* replacement compiler. This strategy is relatively simple because it calls for relatively few changes, because it doesn't require modification of compiler code or application code.

The strategy can be simplified further by utilizing stubs generated for the target CORBA environment. The CORBA stubs are used to convert function calls and returns to IIOP. A DCE-over-CORBA data translation layer (or *adapter*) would then need to be generated for each DCE interface undergoing migration.

These adapters would probably be compatible with a single CORBA product (VisiBroker in this case), though the relevant differences between CORBA products supporting the POA specification and would be few or none. For instance, the label used for a CORBA skeleton may vary between CORBA products, so that adapters would need to be recompiled, but this would not affect source code.

In addition to these stubs, facilities are provided by middleware platforms such as DCE that might need to be provided for migration. If a legacy DCE application makes use of DCE's security, threading, and/or directory services, then a DCE look-alike interface must be provided for whatever replacement services are provided.

24 Unfortunately, OODCE may have commonly been "extended" by EOSDIS code. Such extensions of OODCE may undermine the simplicity of the interface. For example: the `DCEFilePassword` class.

5.2.4 Combining Bridging with Source-Level Interoperability

A source-level interoperability strategy may be coupled with a DCE-CORBA bridge to capitalize on the complementary strengths of each approach.

- **Bridging**
 - Facilitates immediate access to DCE/ODCE servers by CORBA clients.
 - Automatic: doesn't require recompilation of source code.
 - Facilitates creation of new CORBA clients and gradual conversion of legacy clients to CORBA.
- **Source-Level Interoperability**
 - Facilitates the execution of DCE/ODCE clients and servers over CORBA.
 - Static: requires minimal runtime overhead.
 - Facilitates a more permanent migration of ODCE clients and servers to CORBA.

The two approaches can be combined by way of a three-stage transition plan:

- **Start by activating a bridge between the legacy DCE domain and a new CORBA domain.**
- **Use a code migration strategy to gradually move clients to CORBA.**
- **Gradually move outer-tier servers (called only by non-servers) to CORBA. Access these servers directly (via CORBA).**
- **Finally, move remaining (middle-tier and back-end) servers to CORBA.**

5.3 Technical Discussion

5.3.1 Marshalling Strategies

DCE and CORBA both use stubs for marshalling and un-marshalling of remote procedure calls.

Figure 4: CORBA Marshalling

When using SOAP, the strategy is the same, only the transport data packaging (e.g., IIOP) is replaced by SOAP.

The marshalling structure for DCE-to-CORBA migration is the same in principle, however, in practice it may adopt a double-layered stub architecture, because much of the marshalling work can be off-loaded onto CORBA stubs and skeletons.

Figure 5: Dual-Stub DCE/IIOP Marshalling

The advantage to this approach is that it is a little simpler to implement; the disadvantage is that it depends on compatibility with CORBA skeletons (and stubs to a lesser degree) that have historically varied from vendor to vendor and from release to release. Significant changes were encountered when moving from VisiBroker 3.x to 4.0, and minor changes were encountered when moving from VisiBroker 4.0 to 4.1.

On both client and server ends, the function call makes a lateral transition between DCE and CORBA models:

Figure 6: DCE Method Invocation Flow via CORBA

5.4 Design

5.4.1 Requirements

Because the objective is to port OODCE applications without modifying source code, the top-level code remains unchanged, and must interact with a CORBA ORB. Most of the hard work is performed by CORBA skeletons and stubs generated by a CORBA utility such as *idl2cpp*. Before any skeletons or stubs can be generated, OODCE IDL files must be converted to OMG (CORBA) IDL. To enable the OODCE legacy code to communicate with the CORBA skeletons and stubs, an intermediate object layer must be generated from the OODCE IDL files.

In addition:

1. OODCE data types must be mapped to CORBA data types (see *Definition Conversions* in the diagram).
2. Services must be supplied where CORBA doesn't provide an equivalent to an OODCE class (e.g., DCEServer and DCEPthread).

5.4.2 Method Invocation Flow

Figure 7: Invocation Flow Diagram

The flow of a method invocation moves from left to right in the diagram.

Invocation begins in the legacy client program (1), where a method of an OODCE client object (2) is invoked.

The OODCE client object performs the necessary conversions of non-primitive types, represented by the arrow pointing to the CORBA stub object (3).

By way of the CORBA stub object's base class, the invocation is passed across the ORB (4) to the corresponding skeletal base class (5).

The skeletal object (6) built upon the base class then performs the necessary conversions of non-primitive types, represented by the arrow pointing to the OODCE server base class (7) and a special purpose class developed for this project, named *oodceBase*.

Finally, a legacy server implementation object (8) is invoked within the legacy server process.

Return data and [out] parameters are returned along the same path, in the opposite order.

The four columns represent middleware zones (left to right):

- I. OODCE client zone.
- II. CORBA client zone.
- III. CORBA server zone.
- IV. OODCE server zone.

The left-to-right arrows between OODCE and CORBA zones represent the critical type mapping functionality.

5.4.3 The Code Migration Process

Figure 8: Migration Process

The migration process diagram displays legacy code at the top, with VisiBroker utility and output at the bottom, and the contributions of this exercise:

- **Converter**
 - OODCE output
 - OMG IDL output
- **Libraries and Headers**
 - DCEPThread
 - DCEServer
 - TypeMapper
 - Dce_defs.h

The four basic steps illustrated in the diagram are outlined below:

1. The migration process is initiated feeding OODCE IDL into the “Converter” utility.
2. The “Converter” utility generates two types of data:
 - A. OMG IDL.
 - B. OODCE classes & glue.
3. The newly generated OMG IDL is fed into a CORBA IDL compiler (the VisiBroker *idl2cpp* utility).
4. Idl2cpp generates the CORBA stubs and skeletons to be used under the auto-generated OODCE classes and glue.

5.4.4 Migration Process Steps

- 1) Presume there is a pre-existing OODCE application, including an IDL file, headers, stubs, compiler-generated sources, and application sources. Make a copy of the entire file set for the purpose of migration.
- 2) Run the OODCE IDL file through the *Converter* utility that generates the following files:

Table 12: Converter Output

<interface>.idl	OMG (CORBA) IDL file.
<interface>S.H	header file included by legacy server code.
<interface>S.cpp	source file which provides type mapping and other facilities on the server side.
<interface>C.H	header file included by legacy client code.
<interface>C.cpp	source file which provides type mapping and other facilities on the client side.

- 3) Run the new <interface>.idl file through a CORBA IDL compiler (such as *VisiBroker's idl2cpp*). This generates a number of CORBA-style stub and skeleton files which do not need to be modified after they are generated.

- 4) Create a project, using a number of pre-constructed files that provide definitions and functionality:

Table 13: Shared Libraries & Headers

dce_defs.h	CORBA-friendly definitions for DCE types.
unistd.h	an empty placeholder file.
RPCErr.H	an empty placeholder file.
Pthread.H	OODCE-compatible threading definitions.
DCEPthread.lib	OODCE-compatible threading library.
Server.H	DCEServer replacement definitions for main program file.
DCEServer.h	DCEServer replacement definitions for general use.
DCEServer.lib	DCEServer replacement library.
TypeMapper.h	Type mapping facility definitions & declarations.
TypeMapper.cpp	Type mapping facility implementation code.
<interface>.idl	an empty placeholder file.

5.4.5 Code Generation

Source porting adapter code must be generated automatically for this strategy to work.

To minimize the chore of automatic code generation, and to maximize its efficiency, an effort has been made during development of the migration architectures to distinguish interface-specific code from general use code that does not change from interface to interface.

It is important to keep in mind that, though automatic code generation is of critical importance, it must be derived from the architecture it is intended to generate code for and therefore should not be pursued until the form of the code to be generated has been determined. Because code generators were developed early in the process of this investigation, and because a variety of strategies have been explored since, numerous modifications have been made to those generators in the course of this investigation.

5.4.5.1 Stages

The code generation process can be divided into three operations: parsing, analysis, and generation. First, the interface code (OSF IDL in this case) must be parsed. Second, it is useful to analyze the parsed code to simplify the code generation step. This is useful because, as has already been pointed out, code generators can and will change as changes are made to the migration architecture; therefore the code performing the generation should perform as little analysis as possible. Finally, the adapter code is generated. In this case, the code generation utility also generates OMG (CORBA) IDL code to serve as inputs for OMG IDL compilers.

Unlike the project of Schmidt, Gokhale, and Moyer [4.1], the tools developed in support of this thesis have not employed a third-party IDL parser. This was an easy choice to make at the outset with simple interfaces, but as more sophisticated IDL elements were incorporated such as structured typedefs, imports, and a variety of DCE-specific attributes (including several that link function parameters and struct members), the parser had to be restructured. An attempt was made to incorporate the Sun IDL compiler [4.1], which had been discontinued. The compiler code was difficult to find, and once found was difficult to port, so the parser that was developed to support this thesis was retained as the parser of choice for this investigation.

5.4.5.2 Handling the out Parameter Attribute

DCE RPC and IIOP both handle out parameters. This is necessary for supporting any programming language that allows data to be passed back from a procedure by way of a parameter, whether directly or indirectly. It is also useful for facilitating the return of more than one data value.

Care must be taken when designing the code structures that will eventually be generated automatically to ensure that out parameters are called such that the calling procedure can capture changes to their associated data on return. Every out parameter must be converted and marshaled for return. The marshalling is similar with IIOP and SOAP.

In IIOP, out (and inout) parameters are placed in a reply structure as follows:

The IDL declaration:

```
double example (  
    in short m,  
    out string str,  
    inout long p);
```

The reply structure:

```
struct example_reply {
```

```

        double    return_value; // return value
        string    str;
        long      p; // ... to the rightmost
    };

```

5.4.5.3 Conformant Types

The OSF/DCE IDL `size_is` and `max_is` attributes must be identified during code generation to associate struct members or function parameters, so that they can be used together to generate code for conformant arrays and structures.

An IDL example:

```

double example (    [in] handle_t h,
                    [in] short len,
                    [in, size_is(len)] byte bytes );

```

The code for a call to the DCE implementation may be generated as follows:

```

short len;

TypeMapper::to_short(len, SOAPparams[1]);

byte *bytes = new byte[len];

TypeMapper::to_idl_byte_array(bytes, SOAPparams[2],
len);

double return_value =
    example ( h,
              len,
              bytes[] );

```

Note that the above code is generated such that the bytes array is allocated and initialized using the value of `len`.

5.4.5.4 Output

The following files are generated automatically:

- The client adapter files `<interface>C.h` and `<interface>C.cpp`.
- The server adapter files `<interface>S.h` and `<interface>_Mgr.cpp`.
- The OMG IDL file `<interface>.idl`.

5.4.5.5 Limitations

The parser developed in support of this thesis supports a subset of the OSF IDL syntax. This incomplete coverage is largely due to prioritization on other issues that are more fundamental to this investigation, though some IDL syntax is not supported due to the fact that some OSF IDL features are not portable. Of the elements of OSF IDL, the following are not yet supported:

- The interface attributes `endpoint`, `exceptions`, `local`, and `pointer_default`.
- Constants.
- Unions.

- Enumerations.
- Pipes.
- The type attribute `transmit_as`.
- The field attributes `last_is`, `length_is`, `min_is`, `max_is`, and `ignore`.
- Pointer attributes.
- Operation attributes `idempotent`, `broadcast`, `maybe`, and `reflect_deletions`.

5.4.5.6 Constructed Types

Support for translation of basic data types is straightforward. A parameter is recognized as a particular data type during parsing, and then is matched with a type mapping function during code generation. Support for constructed types is not so simple. Code must be generated that defines each constructed type in a format that is directly compatible with the target middleware (CORBA or SOAP).

In the CORBA case, an OMG IDL compiler is employed for the generation of low-level stub code. The compiler generates a class for the constructed type. The compiler developed to support this thesis does the same, except it generates a struct that is functionally identical to that generated by the OSF IDL compiler. Next, a custom translator is generated. This translator calls a predefined translator for each struct member.

5.5 Case Specifics

5.5.1 Simple and Threaded Math Examples

The initial technical objective of this investigation, as mandated by NASA, was port two example programs, one *simple math* program and one *threaded math* program.

These programs consist of the following source files:

- client.C
- server.C
- <interface>_<ver>_Mgr.C
 - math_1_0_Mgr.C
 - matha_1_0_Mgr.C

The math interface is simple enough to be regarded trivial. It depends upon a single data type - double (other than the obligatory `handle_t`), and a single function form:

```
double add (  
    [in] handle_t h,  
    [in] double a,  
    [in] double b );
```

Though the math interface consists of four function signatures, the four functions share a single form:

```
double function( handle_t, double, double );
```

5.5.2 Simple Implementation of the ECS Subscription Server Interface

The second technical objective is to port a trivial implementation of the ECS subscription server interface, without changing any implementation source code.

There are two good reasons for working with a trivial implementation:

- Access to the actual NASA implementation code was denied.
- Using the actual implementation code would introduce issues beyond the scope of this exercise.

The subscription interface is much more complex and therefore realistic than the math interface. The subscription interface consists of 18 functions in 9 distinct forms, contrasted with a mere 4 identical functions in the math interface. By “distinct forms” is meant function signatures that differ by parameter or return types. Parameter types include arrays, pointers, and custom types. Return types include pointers and three different custom types, one of which is a struct.

Most challenging, rather than the mere count of functions and parameter arrangements, is the variety of data types and IDL elements used by the subscription interface:

- Imported interfaces.
- Type definitions.
- Constructed types.
- Pointers.
- Arrays.
- Data handles.
- Context handles.
- Buffers.
- Conformant structures.
- Conformant function parameters.
- DCE strings.
- [out] parameters.
- Returned pointers to structures.

One good example of the added complexity in the subscription interface is the `DistGetClientSubscriptions` function:

```
/* GetSubscriptions( GlClient)
   - get all subscriptions for this user */

[idempotent]
DsTShByteBuffer*
DistGetClientSubscriptions(
    [in] handle_t h,
    [in] DsSdDCESubscrHandle c,
    [in, size_is(len)] byte client[],
    [in] long len
) ;
```

First notice that this function returns a pointer to a buffer. That buffer is a conformant structure. Next, notice the handle type `DsSdDCESubscrHandle`. This is a context handle. Finally, notice how the array parameter `client[]` is linked to the scalar parameter `len`. These two make the parameter list conformant as well.

5.6 Process

5.6.1 Simple and Threaded Math Examples

The interfaces, implementation, client, and server code used in this example were provided by NASA; hence the configuration described herein was indirectly determined by NASA.

The examples were executed in the VisiBroker v.3.3 environment.

The steps of test execution are as follows:

1. Process each interface (IDL) file with the "Converter" utility, which performs two tasks:
 - a. Replaces the *idl++* utility with respect to high-level code generation.
 - b. Converts DCE IDL to OMG IDL.
2. Process the resulting OMG IDL with the *idl2cpp* utility provided with VisiBroker. This step replaces the use of *idl++* with respect to low-level code generation.
3. Create client and server projects consisting of legacy code and the output of steps 1a and 2.
4. Build the client and server executables.
5. Launch the VisiBroker *osagent* and naming service.
6. Launch the server application(s).

7. Launch the client application(s).
8. Record server and client output.

The examples provided for this exercise use two similar interfaces: *math* and *matha*. For the purposes of CORBA, these interfaces are identical. The example is constructed such that a server program may register one of two pairs of objects with the ORB, depending on how that server program is launched. Since the client expects to bind to both object pairs, the server program is launched twice (each launch in a separate process).

Once the server processes are launched, the client program is launched and its output monitored. The objective is to see that the client binds successfully to all server objects, and succeeds in receiving the proper results from each server object's member functions.

5.6.2 The ECS Subscription Server Interface

The implementation, client, and server code used in this example were developed, tested over OODCE and SunOS, and provided by Sandeep Gawai, a graduate student at San Jose State University.

This example was executed in both the VisiBroker v.3.3 and v.4.0 environments.

The basic execution steps for this example are identical with those of the previous (math) example.

5.7 Previous Research

As was stated before in this thesis, a previous attempt was made to develop automation tools for DCE to CORBA migration. This was a collaboration between Washington University and Bellcore (now Telcordia Technologies, Inc.) [4.1]. Unfortunately, only a short paper has been published on this work, and the details have been withheld. Because of this, that previous research was of little value to this investigation.

Among the known differences between this investigation and that of Gokhale, Schmidt, and Moyer are:

- The legacy application addressed in this thesis is based on OODCE. Gokhale, Schmidt, and Moyer made no mention of OODCE.
- This thesis describes a strategy that is extensible from a CORBA based solution to a SOAP based solution.
- Gokhale, Schmidt, and Moyer commented very little on the development of static frameworks and runtime facilities. In contrast, this thesis discusses these components in detail.
- The conversion utility used in this thesis is developed entirely for this exercise, and is not based on any third-party compiler product.
- Gokhale, Schmidt, and Moyer address interoperability problems as issues to be avoided while developing the legacy application. In the case study for this thesis the legacy application had already been developed, so problematic code had to be ported or changed.

Though the circumstances, strategy and developments of Gokhale, Schmidt, and Moyer differ substantially from this investigation, their paper points out a number of pertinent porting issues:

1. CORBA doesn't support idempotent functions. This could be significant, as many EOSDIS functions are idempotent. For instance, 6 of the 18 methods in *DsSdDCESubscrIDL.idl* are idempotent. Fortunately, the idempotent feature is intended primarily for unreliable protocols such as UDP. When using TCP, a client that has received a reply from a server method can be sure it has executed that method exactly once. In this case, method idempotence is much less important.²⁵
2. CORBA doesn't support DCE context handles transparently.²⁶ 14 of the 43 EOSDIS interfaces, including *DsSdDCESubscrIDL.idl*, contain context handles. Each of these interfaces uses a single context handle to define a pointer type that refers to itself. In the case of the subscription server interface, this context handle is either passed to or returned by each of its 18 methods. The purpose of these context handles is to maintain session information between a client and a server. The client holds a context handle to identify itself and its current session with the server, but cannot make any other use of the handle. The server generates the handle, and uses it to store session data. The client and server depend on DCE stubs to manage the context handle, specifically to call a server-implemented routine in the case of a connection failure.²⁷ CORBA does not provide this service, so it must be placed above the skeletons and stubs, yet

²⁵ See Power Programming with RPC; by John Bloomer; pub. O'Reilly, 1992; pages 109-10, 198.

²⁶ Gokhale, Schmidt, & Moyer, page 6, col. 1.

below the legacy application code. To emulate DCE functionality, a way to detect a connection failure must be determined. Perhaps a simple timeout mechanism would be adequate for demonstration purposes. A method could be defined in a base class that could be inherited by the `<interface>_<ver>_Mgr` class in the `<interface>S.H` file, and overridden by manager code. How, then, would that method be called? Every time a server method is called, a threaded timer would be restarted. If that timer reaches a predetermined value, the aforementioned context rundown procedure would be executed.

- a. This timeout management code be declared in the remote methods of the `<interface>_<ver>_Mgr` class in the `<interface>S.H` file, and implemented (defined) in a dedicated source file. The server application code would no longer implement the remote interfaces, but a slight variation of them. This could be facilitated by modifying the CORBA skeletons.
 - b. Instead of having the `<interface>_<ver>_Mgr` class inherit from an automatically generated CORBA skeleton, the `<interface>_<ver>_Mgr` class could contain an instance of a subclass of that skeleton. The contained subclass would, upon being called, call an `<interface>_<ver>_Mgr` method, but it would also contain context management code.
3. Gokhale, Schmidt, and Moyer advise to “avoid the use of pointers in DCE interfaces since CORBA doesn’t provide a mechanism for specifying this in a

²⁷ See Guide to Writing DCE Applications, 2nd Ed.; by John Shirley, Wei Hu, and David Magid; pub. O’Reilly and Associates; pages 240-3.

portable and transparent manner.”²⁸ As has been previously stated, the ECS case does not provide the luxury of avoiding pointers. Pointers and handles are used commonly in the EOSDIS interfaces (in at least 33 idl files and 2 acf files). In the case of the subscription server interface, the pointers (or handles) used are the `DsSdDCESubscrHandle` context handle (18 instances) and the `DsTShStringIDL` string pointer (4 instances).

4. Gokhale, Schmidt, and Moyer advise to “avoid using DCE ACF attributes (e.g., `comm_status` and `fault_status`)”.²⁹ 9 of 11 EOSDIS ACF files use one or more of these attributes. Unfortunately, as with other issues, it’s too late to “avoid using DCE ACF attributes”. Because access to ECS source code was denied by NASA, the extent to which ECS code depends upon ACF attributes cannot be determined.
5. “Be careful of using DCE “varying arrays”, which only pass a part of the array from client to server.”³⁰ One such array appears in a single EOSDIS IDL file. Once again, ECS does not meet the design guidelines proposed by Gokhale, Schmidt, and Moyer, but fortunately ECS does not use DCE varying arrays extensively.

Having reviewed the constraints of the case addressed by Gokhale, Schmidt, and Moyer, it is clear that any migration of a legacy application such as NASA’s ECS cannot work within those constraints. Wherever the model of these researchers has

²⁸ Gokhale, Schmidt, & Moyer, page 6, col. 1.

²⁹ Gokhale, Schmidt, & Moyer, page 5, col. 2.

³⁰ Gokhale, Schmidt, & Moyer, page 6, col. 1.

fallen short with respect to the ECS case, the model must either be extended or supplemented with alternate strategies.

5.8 Generating an OMG Interface Definition File

A modified copy of the `<interface>.idl` file must be generated for CORBA compatibility. This CORBA-compliant *.idl* file will be processed by the *idl2cpp* utility.

5.8.1 Feasibility

Not every DCE IDL element maps to OMG IDL, but the two languages and the middleware capabilities they specify are similar enough to make conversion practical in many cases.

The DCE IDL elements that do not map well to OMG IDL are discussed in the following paragraphs.

The `idempotent` operation attribute is unnecessary in CORBA, because it is irrelevant when using reliable protocols such as TCP/IP.

The `broadcast` operation attribute specifies that a remote procedure call can be broadcast to all DCE hosts on a network, and that the first reply is the one used by the client. Though this feature is not available in CORBA, it is not expected to be vital to the operation of any legacy application. The `broadcast` attribute does not appear to be used by ECS.

DCE clients can maintain state of the server by using context handles. These handles have no equivalent in CORBA. Context handles can be facilitated with fairly straightforward management code. Full functional equivalence may be difficult to achieve over CORBA, because CORBA does not monitor clients in the manner that DCE does.

DCE's specialized attribute configuration file (ACF) attributes have no equivalent in CORBA, and may cause difficulty if used by a legacy application. The exception handling attributes `comm_status` and `fault_status` are two common examples of ACF attributes. This is not necessarily an issue for an IDL converter, because an IDL converter can process DCE IDL without processing ACF's. DCE clients would, however, need to be modified to execute over CORBA. EOS does use ACFs to a limited extent.

CORBA has no feature that maps well to DCE pipes, though OMG Audio/Visual Streaming could perhaps be adapted to implement DCE pipes (this specification is not actually part of CORBA³¹, nor is it implemented in the major ORBs). If an IDL converter does not support pipes, it should throw an exception if it encounters the DCE pipe type in an IDL file.

The `string` and `pointer` attributes can be converted fairly easily.

³¹ OMG Audio/Visual Streams is an OMG Telecommunication Specification. See http://www.omg.org/technology/documents/formal/omg_telecoms_specs.htm

Varying arrays are difficult to emulate over CORBA, so an IDL converter should throw an exception if it encounters varying arrays in an IDL file. Conformant arrays, which use the `size_is` attribute, can be converted.

These limitations help to determine what DCE interfaces can be automatically converted. The presence of pipes, ACFs, varying arrays, and context handles can make automated conversion difficult.

If the frequency of such elements is limited, the automated process can be supplemented by manual conversion. The viability of this strategy must be determined on a case by case basis. In the ECS case, the strategy appears to be feasible.

5.8.2 Developing a Conversion Utility

To achieve the objective of source-level migration, an IDL conversion utility is necessary. Because such a utility is unavailable at this time,³² one must be developed for this project.

Two advantages to creating a custom conversion tool are:

1. Type mappings can be defined precisely, instead of attempting to adjust to the mappings of a pre-existing tool.
2. A custom-made tool can be integrated with other utilities developed to support this thesis.

5.8.3 The Conversion Utility

A conversion utility was successfully developed for this investigation. This fully-automated utility is capable of generating CORBA IDL and OODCE-over-CORBA stubs from legacy DCE IDL without any manual intervention.

³² A partial solution has been developed by Sun Microsystems for OMG. Since it depends upon the development of a code generation back-end to be fully functional, and since it makes significant use of UNIX system calls (the present implementation happens to be on Windows), it has not been utilized in this investigation.

5.8.4 IDL Elements Not Supported

The limitations to a conversion strategy determine what conversion work must be done by other means.

The conversion utility and framework developed for this investigation was enhanced to support the migration of the ECS subscription interface to OODCE-over-CORBA.

Some features used by ECS, but not by the subscription interface, are not supported by this strategy. These features must be converted manually.

DCE features that must be converted manually include the following:

- **ACF attributes**
- **Varying arrays**

Other DCE features that do not appear to be used by ECS are not supported by the present strategy:

- **Broadcast methods**
- **Conformant types based on the `max_is` attribute.**
- **Compound (nested) structures.**

5.9 The Elements of IDL Conversion

The conversion utility developed for this investigation automatically converts most DCE IDL elements into CORBA IDL elements:

1. Removes the global `uuid` attribute and creates an equivalent constant in the interface it identifies.
2. Removes the global `version` attribute and creates an equivalent constant in the interface it describes.
3. Removes all square brackets.
4. Replaces the combined `[in, out]` method attributes with `[inout]`.
1. Changes pointers to single-element CORBA sequences.
2. Changes arrays to unbounded sequences.
3. Changes handles to sequences of sequences.
4. Inserts a `typedef` statement for each sequence type introduced.
5. Changes `import "<";` statements to `#include <>` statements, and moves them out of the interface into global space.
6. Changes occurrences of the DCE `byte` type to the CORBA `octet` type.
7. Changes indeterminate arrays with the `size_is` attribute (called *conformant arrays* in DCE) to CORBA unbounded sequences.
8. Defines `context_handle` as `sequence <octet>`.
9. Removes all occurrences of `[idempotent]`. CORBA does not support it. Though this feature may impact the reliability of remote procedure calls, it does not appear to affect the interface, or even the set of possible outcomes.

10. Simple structs which are introduced only to emulate DCE *conformant arrays* are replaced by unbounded octet sequences (the CORBA equivalent of *conformant arrays*). The original struct is expected to have been designed with specific member types and names.
11. Each reference to a custom type defined in an imported interface is prefixed with “<interface>::”.
12. Each interface definition block is closed with a semicolon.

Other conversions may easily be added to future versions of the utility:

13. Convert *small* and *unsigned small* to *octet*.
14. Convert *hyper* and *unsigned hyper* to *octet* [4.1].

5.10 Other Issues

Visibroker's idl2cpp.exe generates a pair of header files for each interface, even if multiple interfaces share a single module³³. This means that the module is defined in each header file, leading to a multiple definition problem if more than one of these headers is included in a single source file. Because the OODCE examples being tested do just that, and also because the examples don't involve modules anyway, modules are not used in this implementation, though the use of modules will be discussed.

³³ Observed in v3.3; not tested for v4.x.

5.11 The Client

CORBA applications typically instantiate a client object by calling a binding function generated by an IDL compiler. OODCE client applications are more elegant, inasmuch as they instantiate a client object by doing just that: *instantiating* a C++ object from a class generated by `idl++`. This is a convenient distinction when porting from OODCE to CORBA, because it means that a wrapper class can be created over stub code (generated by an IDL compiler) that makes the stub code look just like an OODCE class.

5.11.1 Miscellaneous Details

In the file `<interface>C.H`, the `#includes` must be replaced with the following:

```
#include <dce_defs.h>
#include "<interface>_c.hh" // Interfaces generated by idl2cpp
```

The `dce_defs.h` file contains CORBA-compatible definitions for DCE data types. It is similar to the `idlbase.h` file, but it `#includes` `corba.h` and replaces NDR types with CORBA types. For instance:

```
// typedef ndr_long_float    idl_long_float ; (REPLACED)
typedef CORBA::Double        idl_long_float ;
```

In `<interface>C.H`, add the following data member to the class `<interface>_var`.

```
<module>::<interface>_var theObject;
```

The method declarations themselves remain unmodified.

A corresponding source file is created, which is somewhat arbitrarily named `<interface>C.C`. This file contains constructor implementations which call `CORBA::ORB_init` and `<module>::<interface>::_bind`. According to the CORBA specification (section 4.6) [5.3], there is no problem with calling `CORBA::ORB_init` more than once in the same process.

5.11.2 Basic Form of a Inter-Middleware Invocation

In cases where DCE types map directly to CORBA types (e.g., the math example), a pair of statements is inserted into each interface method that connects the OODCE-alike class to the CORBA stubs underneath:

```
<return type> ret = theObject-><method name>(  
    <parameter 1>,  
    <parameter 2>,  
    ...  
    <parameter n> );  
  
return ret;
```

Member functions that use the `void` return type would of course omit any reference to returned data.

5.11.3 Advanced Type Mapping

In cases where parameter or return types do not map directly (e.g. the subscription interface), the solution gets a bit more complex:

```
<return type> ret = theObject-><method name>(
    <parameter 1 type mapper>(<parameter 1>),
    <parameter 2 type mapper>(<parameter 2>),
    ...
    <parameter n type mapper>(<parameter n>) );

return <return type mapper>(ret);
```

This is as far as a solution has been implemented for this investigation. The problem can become yet more complex where parameters and/or return types are structs or pointers thereto. The likely solution is to recursively call type mappers as needed for each struct member. Though this kind of recursive logic has been utilized in this project, for instance in parsing structured typedefs, recursive logic has not been used for type mapping.

5.12 The Server

In the server case, the OODCE class which server objects are based upon is subject to a similar modification. Similar `#include` replacements are introduced in the

`<interface>S.H` file:

```
#include <dce_defs.h>

#include "<interface>_s.hh" /* Interfaces generated by
                             idl2cpp.exe */
```

Instead of the base class `<interface>_<version>_ABS` inheriting from `DCEObj` and `DCEInterfaceMgr`, it inherits from the `oodceBase` class and links to a subclass of the CORBA skeleton class `_sk_<module>::_sk_ <interface>`.

For example, the CORBA skeleton subclass holds a reference to the OODCE abstract class (a subclass of `oodceBase`):

```
class CORBA_DsSdDCESubscrIDL : public _sk_DsSdDCESubscrIDL
{
private:
    DsSdDCESubscrIDL_1_0_ABS* oodceLayer;
    ...
};
```

Likewise, the `oodceBase` class holds a reference to the base class of the skeleton class (`servantBaseType`):

```

typedef PortableServer_ServantBase* servantBaseType;

class oodceBase
{
protected:
    servantBaseType servant;
...
};

```

As in the client case, the constructor declarations may be altered, but not without making sure the interface is not broken for OODCE applications.

The interfaces and implementations of the server methods (member functions) need not be modified.

5.13 DCEServer

A special class must be developed to emulate the DCEServer class and theServer object used by OODCE server programs. theServer is a Global Server Object (GSO) provided in the OODCE library. To emulate it, an OODCE emulation library must be provided (that contains GSOs such as theServer). An example of such a library has been developed in the course of research for this thesis.

The DCEServer class developed for this investigation provides the following functions, all required by OODCE applications:

```
pthread_addr_t DCEServer::ServerCleanup(void* param);

int DCEServer::RegisterObject(oodceBase& oodceObject);

void DCEServer::Listen(void);
```

These method signatures reveal two details of the implementations.

The ServerCleanup method returns a pthread_addr_t thread pointer. The thread pointer is generated by a thread library, developed for this investigation, that is based on POSIX threads.

The RegisterObject method receives a reference to an oodceBase object. This object is not defined in OODCE, but this is not a problem because oodceBase is

defined in the headers generated by the conversion utility as a base class for the interface stubs that are passed by the legacy application to the `RegisterObject` method.

The `DCEServer` class also contains the following private members:

```
int DCEServer::ServerInit(void);

CORBA::ORB_ptr orb;
CosNaming::NamingContext_var rootContext;
PortableServer::POAManager_var rootManager;
PortableServer::POA_var rootPOA, myPOA;
```

The `ServerInit` method initializes the ORB and the naming service. The remaining members are CORBA-specific variables.

Server applications that use `theServer` access its prototype via the `Server.H` replacement header file, which may be included only once in a given server program. The header containing the actual class details is `DCEServer.h`, and the implementation file is `DCEServer.cpp`.

5.14 Observations

5.14.1 Changes in VisiBroker

With CORBA, the term 'standard' might lead to unrealistic expectations, because the CORBA standard has been a moving target. The VisiBroker product illustrates this issue very well.

The first round of tests for this investigation (of the math examples) in September and October 1999 used VisiBroker v3.3. When testing resumed in April 2000, VisiBroker v.4.0 had become available, so both versions were used in testing.

The most difficult obstacle in upgrading the present solution to support v.4.0 was the task of adopting the POA interface, which should lead to more server portability, but the timing of this upgrade was unfortunate for this study.

A significant, perhaps temporary change, is that the code generated by *idl2cpp* appears to lead to scope ambiguity (the compiler is somehow lead to allow references to data with ambiguous scope) and ambiguity of indirection (the compiler is somehow lead to allow non-pointers to use the 'new' operator as though they are pointers).

Significant revisions have also been introduced to the COS naming service.

5.14.2 Problems with Skeletons and Stubs

5.14.2.1 Returning references (pointers, handles, etc.)

Skeletons and stubs generated by *idl2cpp* do not always contain copies of parameters, but contain only references to parameter data which is passed in as a reference. This is not a problem in and of itself, but it requires discipline in their use.

In the present design, when legacy code returns a data reference, that reference is passed across to skeletal subclass code which passes that reference to a type mapping function. The type mapping function returns its results, often a reference, down to the ORB via its skeleton. Whenever a reference is returned, there is a risk that the data to which it refers may fall out of scope, rendering the reference invalid. Care must be taken to eliminate this risk, perhaps through a careful use of static or member data.

5.14.2.2 Single-Element Sequences

Skeletons and stubs generated by *idl2cpp* contain an apparent bug in their implementation of single-element CORBA sequences: if an attempt is made to use an index other than one on such code, the program will crash with any issuance of an exception, though it appears that an exception is intended by the generated code. This

came to the attention of the author because single-element sequences were used to replace OODCE pointers and handles in this study. Since offsets can be applied to pointers in C++, an attempt was made to use an index on the skeletons and stubs implementing the sequences that represented such pointers. This was a mistake, but one that was made more difficult to recognize by an apparent bug in *idl2cpp*.

5.14.3 Conclusions

The principal of automated source-level migration from OODCE to CORBA is sound, but designing and implementing a complete solution is a challenge. In the course of this investigation, the ECS Subscription interface has been ported to CORBA with a fair amount of difficulty. There has been no attempt to port the entire set of ECS interfaces available to this investigation.

An automated solution has been developed, so that the same work need not be repeated for every interface of every application. The automation tool developed for this investigation generates (1) OMG/CORBA IDL and (2) OODCE-over-CORBA skeletons and stubs.

This solution has been implemented on several combinations of VisiBroker and Windows releases:

- 1. VisiBroker v.3.3 on Windows 98 and 2000**
- 2. VisiBroker v.4.0 on Windows 98 and 2000**
- 3. VisiBroker v.4.5 on Windows 2000**

With regard to migrating more realistic examples of legacy code, Gokhale, Schmidt, and Moyer state the following:

It is hard to port from DCE to CORBA since many features do not map directly. Therefore, to achieve some degree of portability to transition from DCE to CORBA it is necessary to avoid certain DCE features. In this context, portability focuses on writing DCE applications that may some day need to port to CORBA.³⁴

This is, of course, an unlikely scenario. Most real legacy DCE and OODCE code could not have been designed with CORBA-friendly constraints. The EOSDIS project is no exception. The EOSDIS IDL and ACF files are packed with non-portable and semi-portable elements.

³⁴ Gokhale, Schmidt, & Moyer, page 5, col. 2.

Though an application that uses the OODCE interface layer may be portable insofar as the OODCE model is portable, uncertainties remain because the EOSDIS project extends (and undermines) the OODCE interface layer with DCE-based implementations. This means that simplifying assumptions that have been made in the design of this migration strategy are not entirely valid. Rather, the DCE-to-CORBA work of Washington University and Bellcore may prove more applicable, though it is known that their strategy makes assumptions that do not apply to the EOSDIS environment. In any case, the applicability of the Washington University/Bellcore solution cannot be known because the details are not available to the public.

Though problems exist with the source-level migration strategy employed by both projects, one may be well-suited as a complement to a bridging strategy. Source-level migration could be applied where the legacy code passes a portability test, and then bridging might be used where the source code itself cannot be ported in this manner.

5.14.4 Minor Problems

It appears that CORBA *modules* cannot be implemented at this time³⁵ because VisiBroker's `idl2cpp` appears to create multiple definition conflicts when a module contains multiple interfaces (see section on `idl`). This does not hinder source-level migration in the ECS case.

³⁵ VisiBroker 4.0

6 RPC over IIOP

6.1 Introduction

Rather than using CORBA skeletons and stubs to translate message invocations and returns to and from IIOP, the interface-specific adapters could be designed to perform the translation themselves, probably with the assistance of an IIOP library, which would need to be developed.

There are two general ways to implement OSF RPC over IIOP:

- RPC-API over IIOP: Re-implement the OSF RPC API to use IIOP rather than OSF RPC.
- RPC Stubs over IIOP: Re-implement the high-level interfaces generated by the IDL (or IDL++) compiler.

6.2 RPC-API over IIOP

To understand the feasibility of RPC-API over IIOP, it is necessary to understand the structure of an RPC call.

The structure of an RPC call will be discussed from two perspectives: first DCE, then OODCE.

6.2.1 DCE

Every RPC call begins when a client program calls a function in a stub file. Within the function, a number of RPC library functions and macros are executed. A total of 16 such actions are performed in the client stub, some of them conditionally. The basic tasks these actions represent are as follows:

- Initialize the client.
- Temporarily disable asynchronous threads.
- Initialize marshalling.
- Begin the call.
- Marshal function inputs.
- Call the function.

- **Un-marshal function outputs.**
- **Perform various clean-up tasks.**

On the server end consists of 20 actions. The basic tasks these actions represent are as follows:

- **Initialize the server.**
- **Initialize marshalling.**
- **Un-marshal function inputs.**
- **Disable asynchronous threading.**
- **Enable general threading.**
- **Call the implementation code.**
- **Marshal function outputs.**
- **Transmit RPC function return.**
- **Perform various clean-up tasks.**

The following tables illustrate this process in technical detail.

6.2.1.1 DCE Process Tables

The following two tables outline the DCE client/server RPC process for a simple function call. The tables are based on actual code output from IDL++. The parameters and return of which share a single basic IDL data type (double).

6.2.1.2 Client

Table 14: DCE RPC Stage 1 of 2

	Action (function or macro)³⁶	Remark
1	RPC_SS_INIT_CLIENT	if(!rpc_ss_client_is_set_up) rpc_ss_init_client_once();
2	RPC_SS_THREADS_DISABLE_ASYNC	state=pthread_setasynccancel(CANCEL_OFF)
3	rpc_ss_init_marsh_state	
4	TRY	
5	rpc_call_start	
6	rpc_ss_ndr_marsh_interp	
7	rpc_call_transceive	Execute RPC over network; see next stage
8	rpc_ss_ndr_unmar_interp	
9	CATCH	
10	FINALLY	
11	rpc_ss_ndr_clean_up	
12	rpc_ss_call_end_2	
13	rpc_ss_mem_free	
14	rpc_ss_report_error_2	
15	RPC_SS_THREADS_RESTORE_ASYNC	pthread_setasynccancel(state)
16	ENDTRY	

³⁶ From the function op0_csr, in the file <iface>_cstub.c.

6.2.1.3 Server

Table 15: DCE RPC Stage 2 of 2

	Action (function or macro)³⁷	Remark
1	RPC_SS_INIT_SERVER	if(!rpc_ss_server_is_set_up) rpc_ss_init_server_once();
2	rpc_ss_init_marsh_state	
3	TRY	
4	rpc_ss_ndr_unmar_interp	
5	RPC_SS_THREADS_DISABLE_ASYNC	Not in OODCE server stubs. state=pthread_setasynccancel(CANCEL_OFF)
6	RPC_SS_THREADS_ENABLE_GENERAL	state=pthread_setcancel(CANCEL_ON)
7	<iface>_v1_0_epv_t.<fcn#0>	Implementation code
8	RPC_SS_THREADS_RESTORE_GENERAL	state=pthread_setcancel(state)
9	RPC_SS_THREADS_RESTORE_ASYNC	Not in OODCE server stubs. pthread_setasynccancel(state)
10	rpc_ss_ndr_marsh_interp	
11	rpc_call_transmit	
12	CATCH_ALL	
13	RPC_SS_THREADS_RESTORE_GENERAL	Not in OODCE server stubs.
14	RPC_SS_THREADS_RESTORE_ASYNC	Not in OODCE server stubs.
15	rpc_ss_ndr_clean_up	
16	RPC_SS_EXC_MATCHES	exc_matches
17	rpc_ss_send_server_exception_2	
18	ENDTRY	
19	rpc_ss_mem_free	
20	rpc_ss_send_server_exception	

³⁷ From the function op0_ssr, in the file <iface>_sstub.c.

This description, though simplified by means of removing references to numerous DCE-specific data types and constants, reveals that DCE RPC stubs are not abstracted for ease of use or migration, but rather they reveal the inner-workings of DCE to a great extent.

As a matter of course, the intent of this investigation is to study the RPC functions and macros called in this process. However, documentation of DCE RPC at this low level is not readily available. The most complete book on DCE programming, Guide to Writing DCE Applications [1.2], does not cover any of the functions and macros detailed here.

Information on RPC is scarce. Definitions for most RPC macros are readily available on the Internet (see process table columns for definitions), but nothing definitive is available on the RPC functions themselves. Only five references were found in the course of this investigation to any of the RPC functions used here.

Re-implementing the DCE RPC interface would be a laborious task to be sure, but would it be impractical? The RPC interface supplies enough special data types and constants that it is reasonable to think that specifics needed for IIOP could easily be concealed behind those RPC-specific terms. It would likely be a complex translation, but how complex?

Unfortunately, and as implied above, there's more to porting an interface than porting functions and macros. If auto-generated stub code is to be ported to IIOP, then value assignments as a whole must have meaning in an IIOP context, and furthermore no such assignment can be permitted to conflict with and IIOP implementation.

The following code represents the first lines of the `op0_ssr` function in `<iface>_sstub.c`:

```
IDL_ms_t IDL_ms;

volatile ndr_boolean IDL_manager_entered = ndr_false;

volatile RPC_SS_THREADS_CANCEL_STATE_T
IDL_async_cancel_state;

volatile RPC_SS_THREADS_CANCEL_STATE_T
IDL_general_cancel_state;

idl_byte IDL_stack_packet[IDL_STACK_PACKET_SIZE];

rpc_void_p_t IDL_param_vec[4];

idl_long_float a;

idl_long_float b;

idl_long_float IDL_function_result;


RPC_SS_INIT_SERVER
```

```

rpc_ss_init_marsh_state(IDL_type_vec, &IDL_ms);

IDL_ms.IDL_stack_packet_status =
IDL_stack_packet_unused_k;
IDL_ms.IDL_stack_packet_addr = IDL_stack_packet;

TRY

IDL_ms.IDL_offset_vec = IDL_offset_vec;
IDL_ms.IDL_rtn_vec = IDL_rtn_vec;
IDL_ms.IDL_call_h = (volatile
rpc_call_handle_t)IDL_call_h;
IDL_ms.IDL_drep = *IDL_drep_p;
IDL_ms.IDL_elt_p = IDL_elt_p;

IDL_param_vec[0] =
(rpc_void_p_t)&IDL_function_result;
IDL_param_vec[1] = (rpc_void_p_t)&h;
IDL_param_vec[2] = (rpc_void_p_t)&a;
IDL_param_vec[3] = (rpc_void_p_t)&b;

IDL_ms.IDL_param_vec = IDL_param_vec;

```

```
IDL_ms.IDL_side = IDL_server_side_k;
```

```
IDL_ms.IDL_language = IDL_lang_c_k;
```

```
rpc_ss_ndr_unmar_interp(
```

```
    2,
```

```
    160,
```

```
    IDL_param_vec, &IDL_ms);
```

Note that immediately after server initialization, the variables `IDL_type_vec` and `IDL_ms` are passed to the `rpc_ss_init_marsh_state` function.

`IDL_type_vec` is a global block of binary data³⁸, ostensibly containing IDL type data for the interface. `IDL_ms` is an empty structure to be initialized by the function. At this point, the challenge for an RPC-over-IIOP port would be to interpret `IDL_type_vec`.

Next the internals of `IDL_ms` exposed as it is loaded with all the critical settings that will be needed by the `rpc_ss_ndr_unmar_interp` function. Not all of the data may be meaningful to IIOP, but struct fields and constants could conceivably be redefined to suit IIOP. As it appears, it seems plausible that `IDL_ms` would be adequate for IIOP's needs, but were more complex data types to be involved, it seems that IIOP would have to be told more. The most critical data appears to be encoded in `IDL_type_vec`, which cannot be redefined, but that doesn't mean it cannot be reinterpreted.

The next question is how is `IDL_type_vec` formatted? Is an answer to this question published?

³⁸ 194 bytes for an interface consisting of 4 simple arithmetic functions.

In any case, the task does not appear to be simple, and there are no guarantees, even if all pertinent stub design information is published, that IIOP could operate fully - or even adequately - through the DCE RPC stub interface.

At this point in the discussion, the Java RMI interface should be offered up for comparison. The RMI interface has been placed atop IIOP by Sun-JavaSoft, at some cost in regard to functionality. This lends some hope to the objective of developing an RPC-over-IIOP, however, RMI is a much higher level interface than DCE RPC, and RMI's interface involves a much narrower range of functionality than DCE RPC. DCE applications must be expected to access a suite of services. Furthermore, RMI was designed with the IIOP legacy in mind, whereas the design of DCE preceded distributed object technology.³⁹

Schmidt, Gokhale, and Moyer submitted a report of qualified success in porting DCE application source code over CORBA skeletons and stubs, so to the extent that they succeeded, they demonstrated that DCE code could operate over IIOP (via CORBA itself), but they were not porting the interfaces of DCE itself:

³⁹ DCE has object-oriented characteristics, but has less in common with the CORBA object paradigm than RMI.

The work presented in this paper uses the source-level interoperability model. ...our tool generates code that integrates existing DCE-based code with the stubs and skeletons generated by CORBA IDL compilers.⁴⁰

The strategy that Schmidt, Gokhale, and Moyer present sheds light on this topic in other ways:

1. Many of the limitations they revealed to be inherent in their strategy may also affect a lower level DCE RPC migration. These limitations are shown to be significant elsewhere in this report.
2. The higher-level "source-level level interoperability" approach raises a question as to whether re-implementing the DCE RPC interface is any more useful, or as workable.

In regard to the latter issue, it should be clear from the coverage herein of both approaches that the high-level option is simpler and more workable. Data types can be converted at a high level as efficiently as at a low level. In the high-level case, data types are converted between CORBA skeletons/stubs and DCE application code. In the low-level case, data types are converted between the IIOP specification and the DCE

⁴⁰ Schmidt, Gokhale, and Moyer, page 2, column 1.

RPC interface layer. The former requires a few conversions, but the latter requires the development of an IIOP-based ORB with a DCE RPC front-end.

The question of usefulness must also be addressed. The idea of an IIOP-based ORB with a DCE RPC front-end sounds more appealing, at least in principal, than a source-level interoperability solution consisting of IDL compilers and glue-layer generators. In reality, would such an ORB be more useful?

The best target for a source-level interoperability solution, such as that proposed and developed by Schmidt, Gokhale, and Moyer, is a DCE application with a need to be migrated incrementally to CORBA. Beginning with the legacy DCE services, a CORBA ORB would be introduced and servers and clients migrated across to the ORB, one by one. In the case of the low-level approach, a specialized ORB would need to be developed, rather than using an established, COTS ORB. DCE clients and servers could then be migrated one by one, or in groups, as prerequisite services become available for the new ORB. It seems that both strategies cater to similar potential customers:

- Both require recompilation, due to similar dependencies on replacement header files.
- Both require replacements for legacy DCE services.

The critical difference between the two approaches is that the low-level option requires the development of an ORB, or at least a sophisticated adapter to a particular ORB (such an adapter is unlikely to be portable). In the former case, an ORB must be maintained. In the latter case, compatibility with an existing ORB must be maintained.

6.2.2 OODCE

The core issues that apply to implementing RPC-over-IIOP for DCE application migration also apply to OODCE application migration, only more so.

As can be seen from the four process tables on the following pages, OODCE involves more calls to the DCE RPC interface (see tables II and III). This points to the fact that not all DCE RPC stubs are alike, and consideration must be made for variations between stubs associated with different interfaces and IDL compilers.

In addition, OODCE introduces additional data types, constants, and services that must be replaced (see tables 16 and 19).

One advantage of this strategy specific to legacy OODCE applications is that OODCE application developers often modify client and server stubs⁴¹, as well as making direct calls to DCE either when writing application code or sub-classing standard OODCE facilities.⁴² When DCE or OODCE-specific changes are made to standard facilities or auto-generated code, migration cannot be a simple matter of

⁴¹ For an example, see Distributed Objects, by Naveen Hota, pages 27-28

⁴² For an example, see Code Walkthrough, by Frank DeLuca, pages 10-11.

generic replacement. This RPC-over-IIOP solution offers a few extra benefits in this regard because it involves re-implementing a number of DCE functions.

6.2.2.1 OODCE Process Tables

The OODCE client-server RPC process for a simple function call. The following tables are based on actual code output from IDL++. The parameters and return of which share a single basic IDL data type (double).

The four tables represent four stub files and four stages of RPC execution.

- The outer stages (tables 16 & 19) are OODCE stages.
- The inner stages (tables 17-18) are DCE stages.

6.2.2.2 Client

Table 16: OODCE RPC Stage 1 of 4

	Action (function or macro)⁴³	Remark
1	BindInterface	
2	ResetSecurity	
3	TRY	
4	op0_csr	DCE client stub - next stage
5	CATCH	
6	TRY	Retry (6-11) if rebinding
7	RebindInterface	
8	Op0_csr	DCE client stub - next stage
9	CATCH	
10	CATCH_ALL	
11	ENDTRY	
12	CATCH_ALL	
13	THROW	
14	ENDTRY	

⁴³ From the function <iface>_l_0::<fcn#0>, in the file <iface>C.C.

Table 17: OODCE RPC Stage 2 of 4

	Action (function or macro) ⁴⁴	Remark
1	RPC_SS_INIT_CLIENT	
2	RPC_SS_THREADS_DISABLE_ASYNC	
3	rpc_ss_init_marsh_state	
4	TRY	
5	rpc_call_start	
6	rpc_ss_ndr_marsh_interp	
7	rpc_call_transceive	Execute RPC over network; see next stage
8	rpc_ss_ndr_unmar_interp	
9	CATCH	
10	FINALLY	
11	rpc_ss_ndr_clean_up	
12	rpc_ss_call_end_2	
13	rpc_ss_mem_free	
14	rpc_ss_report_error_2	
15	RPC_SS_THREADS_RESTORE_ASYNC	
16	ENDTRY	

⁴⁴ From the function op0_csr, in the file <iface>_cstub.c.

6.2.2.3 Server

Table 18: OODCE RPC Stage 3 of 4

	Action (function or macro) ⁴⁵	Remark
1	RPC_SS_INIT_SERVER	
2	rpc_ss_init_marsh_state	
3	TRY	
4	rpc_ss_ndr_unmar_interp	
5	RPC_SS_THREADS_ENABLE_GENERAL	
6	<iface>_v1_0_mgr._l_0_<fcn#0>	OODCE method (stub); see next stage
7	rpc_init_mp	Actions 7-12 are unique to OODCE server stubs.
8	rpc_marshall_ulong_int	
9	rpc_advance_mp	
10	rpc_marshall_ulong_int	
11	rpc_call_transmit_fault	
12	rpc_ss_mem_free	
13	RPC_SS_THREADS_RESTORE_GENERAL	
14	rpc_ss_ndr_marsh_interp	
15	rpc_call_transmit	
16	CATCH_ALL	
17	rpc_ss_ndr_clean_up	
18	RPC_SS_EXC_MATCHES	
19	rpc_ss_send_server_exception_2	
20	ENDTRY	
21	rpc_ss_mem_free	
22	rpc_ss_send_server_exception	

⁴⁵ From the function op0_ssr, in the file <iface>_sstub.c.

Table 19: OODCE RPC Stage 4 of 4

	Action (function or macro)⁴⁶	Remark
1	DCEClientInfo	
2	rpc_binding_inq_object	
3	DCEInterfaceMap::GetObject	
4	DCEInterfaceMap::ActivateObject	
5	<iface>_1_0_ABS::GetRefMon	
6	DCERefMon::CheckAuth	
7	theClientInfoStore.Set	
8	<interface>_1_0_ABS::IncActivity	
9	<iface>_1_0_ABS::<fcn#0>	Execute implementation code
10	<iface>_1_0_ABS::DecActivity	

⁴⁶ From the function <iface>_v1_0_mgr._1_0_<fcn#0>, in the file <iface>E.C.

6.2.3 Conclusions

The RPC-API over IIOP strategy to migrating DCE and OODCE applications to a more sophisticated remote procedure call mechanism may be viable in a technical sense, but this is by no means certain. The resolution of this question depends on further study of DCE internals, if available documentation for those internals is adequate for the task.

Beyond the basics of call-level interoperability, there are the legacy DCE services that would need to be replaced by IIOP-based equivalents. The same issue exists for CORBA-based strategies such as the OODCE migration solution of this thesis and the DCE migration solution of Schmidt, Gokhale, and Moyer.

These different strategies also share the problem that, at a certain point, CORBA and DCE can only be made to share a subset of their respective capabilities.

In pragmatic terms, there is doubt as to whether the RPC-API over IIOP approach offers functionality, performance, or usability benefits beyond those offered by other migration strategies, yet it remains a comparatively challenging task.

The primary benefit of RPC-over-IIOP would be, if it proves technically viable, that it would provide a CORBA-interoperable runtime platform for DCE applications. This is not to say that RPC-over-IIOP could be binary-compatible with legacy DCE applications. Rather, DCE applications would need to be recompiled, re-linked, and re-deployed.

6.3 RPC Stubs over IIOP

Rather than re-implementing the OSF RPC application programming interface (API), one might choose to re-implement the DCE or OODCE stubs generated by the respective IDL compiler.

This approach offers the advantage of not needing to satisfy every detail of the OSF RPC API. However, every API function that is called by a legacy application must be re-implemented for migration, so this alternative does not completely abrogate the need to re-implement the RPC API. Even so, it is a more practical approach for most scenarios, especially those wherein only a few RPC API calls are made by application code.

The primary strength of the general RPC over IIOP strategy is that it is based on a standard protocol, and would not vary by vendor. However, the portability gained by using this approach (rather than a strategy based on CORBA stubs) would not be great, as the portability achieved by utilizing a commercial CORBA product as an IIOP implementation is adequate.

Inasmuch as this alternative only targets IIOP, it does not provide a broker, name service, or transport solution, though it may be rendered interoperable with any

one of the ORBs using IIOP as an internal protocol, and the transport technology (TCP/IP) is easily implemented. This alternative also lacks support for CORBA services.

This approach is deserving of consideration, but it is labor intensive because it requires that most everything beneath the application code be developed. It seems doubtful that a DCE application manager would choose to develop a new IIOP-based middleware when several mature commercial products are available.

Finally, investing in IIOP as a standard protocol, when IIOP's status as a standard may depend on the success of CORBA, may be a questionable undertaking.

[CORBA] “proved too difficult to use in many instances, and too complex and expensive to implement and maintain. Although parts of its core technologies undoubtedly will remain vital to many distributed object deployments, and many products will continue to support CORBA in some form, it has likely hit its high-water mark as a separate, integrated model.” [5.5]

7 Simple Object Access Protocol (SOAP)

7.1 Objective

The objective of this section is to determine the feasibility of implementing OODCE client/server applications over SOAP.

The implementation that has been developed in pursuit of this objective is a simple OODCE application based on a subscription server interface provided by NASA's "Earth Observing System" (EOS) project⁴⁷.

The application is implemented on Windows 2000 for convenience⁴⁸, as well as to demonstrate portability from the UNIX-based infrastructure used by EOS, and is based on an OODCE server that runs over a SOAP-based migration layer that does not require source code changes. This migration layer sits atop a transport layer that utilizes HTTP, CORBA and DCE as XML transport mechanisms.

⁴⁷ <http://eosps0.gsfc.nasa.gov/>

⁴⁸ DCE proved to be much easier to administrate on the Windows 2000 platform than on Solaris. In addition, the version of IBM DCE (1.1) in the lab used by this study is not compatible with more recent releases of Solaris (6-8).

The application also includes three OODCE/XML clients: one that operates over HTTP, another that operates over CORBA (VisiBroker 4.1) and another that operates over DCE (stubs generated by MIDL in OSF compatibility mode).

7.2 The Architecture

In the case of SOAP, the target data format is XML rather than IIOP's CDR. DCE stubs must be replaced by a data translation layer similar to the one generated for migration to CORBA, but in this case the translation layer marshals the function call into the message format it will be transported in (SOAP), rather than depending on CORBA stubs and skeletons.

In addition to this translation, a router-brokerage facility must be developed to direct SOAP commands to their intended server objects. A transport facility must also be provided, though in the case of SOAP this is an easy task, as SOAP can be transported over a variety of mechanisms, the favorite being HTTP, but others such as CORBA, ONC/RPC, and DCE are perfectly feasible.

7.2.1 DCE Method Invocation Flow via SOAP

A DCE function invocation, when implemented over SOAP, undergoes the following basic processes:

- **Marshaling (parameter conversion)**
- **Transport**
- **Routing (server location) ... and further transport**
- **Brokering (server selection)**
- **Unmarshaling (parameter conversion)**
- **Execution**

The same DCE function must undergo the following subset of processes on return:

- **Marshaling (return value and out-parameter conversion)**
- **Transport**
- **Unmarshaling (return value and out-parameter conversion)**

The following two diagrams illustrate the process of DCE method invocation flow by way of SOAP, selecting HTTP as a transport mechanism.

Figure 9: DCE Method Invocation Flow via SOAP

In the diagram above, the client and server are displayed at the top level. Automatically generated code is placed at the next level down, in the stub layer. Just below that is the stub base code layer, which is static code that does not vary by interface. At the bottom is the transport layer, here using HTTP as a transport mechanism.

Figure 10: DCE Method Invocation Flow via SOAP (detail)

In the diagram above, the client and server are represented by bold-line boxes. Automatically generated code is represented by lightly shaded boxes, and is labeled “auto-generated”. The progress of the method invocation from client to server is

portrayed in eleven steps. Steps 2, and 8-10 are performed by automatically generated code.

The steps in the invocation process are numbered in the diagram from one to eleven, proceeding from the upper left downward.

- (1) Legacy DCE client code calls a DCE/SOAP client stub. This stub resembles a DCE client stub to the application code, but rather than marshalling the call for DCE RPC, it marshals the call in the SOAP format. These stubs are generated automatically because they are interface specific.
- (2) A TypeMapper class converts function parameters from DCE types to XML types.
- (3) The general-purpose base class of the client stub invokes a transport object, which may be an HTTP client, or a simple CORBA or DCE client stub.
- (4) The transport object sends the XML stream to a server host, or as will soon be explained, a domain hub that forwards invocations to multiple server-hosts. In this diagram, the invocation is received by a simple server stub, or a skeleton in the CORBA case.
- (5) The invocation is processed by a command broker.
- (6) The command broker receives a single XML string, and extracts an object identifier from that string.
- (7) The command broker searches a table of base class instances for a servant (oodceBase) object that matches the identifier.

- (8) The servant's abstract class, auto-generated to resemble the corresponding OODCE abstract server stub class, executes an auto-generated XML translation class.
- (9) The XML translation class converts the SOAP function parameters into DCE data types.
- (10) The XML translation class then un-marshals the SOAP command into the form of the OODCE manager (_MGR) interface specified in the SOAP header.
- (11) The invocation of the manager interface invokes the corresponding legacy server implementation code.

The structure of the invocation path is primarily linear. The class hierarchies of the OODCE stub models, on both the client and server ends, are represented as stacked boxes, and the translation code is depicted orthogonally, so to be distinguishable from the passage of the invocation itself.

The XML translation class could be merged into the abstract OODCE base class or the manager class, but it has been implemented as a separate class for convenience. The SOAP command could be converted to a method invocation within the abstract base class, but methods called would not be abstract because they would perform XML-to-DCE translation, necessitating two groups of identically named methods within the same class. This would be confusing, so a separate translation class was introduced.

7.2.2 SOAP Routing

DCE client programs use UUIDs to uniquely identify server interfaces and objects. SOAP provides no mechanism for utilizing these identifiers, so one must be developed, depending on the needs of the application.

SOAP clients may be presumed, for the sake of this discussion, to use a single HTTP server to receive all SOAP messages. This HTTP server is analogous to an ORB in a CORBA environment. Behind the HTTP service, a facility must be provided that associates UUIDs with network nodes. This facility herein termed the *command router*. This router directs each SOAP message to a specific server host, which in turn also has an HTTP front end. Each server host maintains a *command broker* facility behind its HTTP service. This broker associates UUIDs with interfaces and objects.

Figure 11: Router-Broker Architecture

The command router facilitates scalability. For simple systems, the command router may be omitted. For large-scale systems, multiple command routers may be employed. For the example implementation, a single command broker was employed, and so the command router was omitted.

Building a brokering architecture atop DCE or an ORB is redundant. Alternatively, SOAP clients could act similarly to conventional DCE clients, obtaining server handles via SOAP, and passing those handles to a SOAP command router that simply passes each handle on to DCE with the method name and parameters. This would of course require that a SOAP interface be developed for the DCE RPC API, but it would abrogate the need for a SOAP-specific routing and brokering architecture.

SOAP applications may be driven over heterogeneous transports, so a simple router-broker facility that doesn't depend on DCE or CORBA has value. The router-broker architecture proposed by this thesis is especially useful for HTTP-based transport, but can be used with CORBA or DCE server objects that act as routers or brokers. Even better, a generic SOAP router-broker architecture can direct invocations to both CORBA and DCE servers, though this has not been attempted in the course of this investigation.

7.2.3 A SOAP Broker Example

The example DCE/SOAP implementation demonstrates how this command broker abstraction can be helpful. Legacy DCE server code is linked to a DCE/SOAP server stub that communicates with a CORBA-based command broker servant. Though this servant is an interface implementation based on a CORBA skeleton, it is also called by an HTTP server and a DCE-based interface implementation. The result is that any DCE server linked to this DCE/SOAP infrastructure is simultaneously available over HTTP, CORBA and DCE.

Figure 12: Multiple SOAP Transports

Little or no data translation is necessary between HTTP and the CORBA and DCE CommandBroker implementations because both use character string data for input and return data.

7.3 Design

7.3.1 Code Generation

Code generation for OODCE-over-SOAP is similar to that of OODCE-over-CORBA (see sections 5.4 and 5.8). This section will focus on the differences.

7.3.1.1 The out Parameter Attribute

SOAP RPC handles out parameters similarly to DCE RPC and IIOP. As was described in the CORBA section of this thesis, the IIOP return structure resembles the following:

```
struct example_reply {  
    double    return_value; // return value  
    string    str;  
    long      p; // ... to the rightmost  
};
```

The SOAP representation is similar:

```
<m:example xmlns:m="Some-URI">
  <return_value>return_value</return_value>
  <str>str</str>
  <p>p</p>
</m:example>
```

This is one area where OODCE-over-CORBA is simpler to implement than OODCE-over-SOAP. This is because CORBA enables out parameter return to occur without much effort on the part of the translator, because C and C++ procedure calls natively support the return of data by way of parameters.

7.3.1.2 Conformant Types

When a client or server stub receives conformant data within a SOAP message, it will need to know what value acts as a limit on what other value.

Though a mechanism that communicates IDL conformance data to the stubs may be necessary in a strict sense, it is not as useful with SOAP as it is with DCE. The SOAP message will by its nature reveal the size of all arrays, so conformant arrays are automatically supported. However, there may be cases where the data transmitted is not as large as the *capacity* of the array. In these cases, the mechanism described for OODCE-over-CORBA would be helpful.

7.3.1.3 Constructed Types

A similar strategy is employed for constructed types over SOAP as has been employed for constructed types over CORBA. However, a class is not generated to represent the constructed data type in SOAP format because the C data type used for all SOAP data is the same (`unsigned short`)⁴⁹. As with CORBA, custom translators are generated that in turn call a predefined translator for each struct member.

⁴⁹ Unicode representation is used in this implementation, though other representations such as ASCII are supported by the SOAP specification.

7.3.1.4 Byte Allignment

Any SOAP implementation has to take byte alignment, or “endianness”, into consideration where binary data types are XML-encoded for transmission via SOAP.

7.3.1.5 Unicode Text

For this project, unicode was originally considered to be a requirement for XML encoding. This requirement has led to more effort in the process of developing and debugging the project code. The choice as to which encoding is best may depend most on the nature of the legacy application.

7.3.1.6 Output

The files generated automatically in the OODCE-over-SOAP case are the following:

- The client adapter files:
 - `<interface>C.h` client header
 - `<interface>C.cpp` client stub
- The server adapter files:
 - `<interface>S.h` server header
 - `<interface>_XML.cpp` XML/C++ type translation
 - `<interface>_ABS.cpp` interface to command broker

7.3.2 Component Diagrams

The text diagrams that follow show the classes on the server side of the OODCE-over-XML architecture proposed by this thesis. Entities in **bold** are concrete classes developed for this architecture. Entities in *italic* are abstract classes developed for this architecture. Entities including *<iface>* and in some cases *<ver>* must be generated for each interface and version respectively.

7.3.2.1 The OODCE/XML Server

Runtime Support Libraries:

```
DCEServer
    manages HTTP;
        calls CommandBrokerImpl;
    initializes DCE and CORBA ORB;
```

The DCEServer class simulates the class of the same name in the OODCE runtime environment. The custom implementation developed for this project manages the initialization of HTTP services, the CORBA ORB, and DCE for the server program that links to it. It also includes the code that is called whenever an HTTP request is received.

```
DCEPThread
```

The `DCEPThread` class simulates the class of the same name in the OODCE runtime environment. The implementation developed for this project is based on the MFC `CWinThread` class.

OODCE Interface and Data Translation Layers:

```
<iface>_<ver>_Mgr
  is-a <iface>_<ver>_ABS
  {
    calls <iface>_<ver>_XML;
    is-a oodceBase;
  }
```

As specified by OODCE, the *Mgr* class is a subclass of the *ABS* class. The *oodceBase* base class is introduced to define all general characteristics of the *ABS* class (that don't vary from interface to interface).

The related *XML* class is introduced for convenience. It plays the role of an intermediate stub layer, and performs all parameter and return type conversions.

```
<iface>_<ver>_XML
{
  calls
  {
    <iface>_<ver>_Mgr;
    TypeMapper;
  }
}
```

Command Parser and Broker Layer:

```
XmlParser
{
  cont
  {
    XERCES::SAXParser;
    MemParseHandlers
  }
```

```

        is-a XERCES::HandlerBase;
    }
}

SoapCommand
{
    contains { SoapParamBlock cont SoapParam; }

    is-a SoapMessage;
}

SoapResponse is-a SoapMessage;

CommandBrokerImpl
{
    is-a POA_CommandBroker;

    calls
    {
        XmlParser;
        SoapCommand;
        oodeBase;
        SoapResponse;
    }
}

DCE CommandBroker
    calls CommandBrokerImpl

```

All three transports (HTTP, DCE, and CORBA) can in fact be used concurrently using this architecture.

The **CommandBrokerImpl** is the hub through which all commands are piped, so that SOAP message strings, whether they are transmitted over HTTP, DCE, or CORBA, pass through **CommandBrokerImpl** to access SOAP-accessible servant objects.

The `CommandBrokerImpl` is accessed via HTTP by means of a direct invocation from the `DCEServer` objects HTTP server component.

The `CommandBrokerImpl` is accessed via DCE by means of a direct invocation from the DCE `CommandBroker` stub. This `CommandBroker` is based on a trivial DCE IDL that defines a single function that receives a string and returns a string.

The `CommandBrokerImpl` is accessed via CORBA by means of polymorphism, as is common in CORBA implementations. This is achieved by declaring `CommandBrokerImpl` a subclass of a CORBA `POA_CommandBroker` skeleton. This `CommandBroker` skeleton is based on a trivial CORBA IDL, equivalent to the DCE IDL described above, that defines a single function that receives a string and returns a string. This strategy is not dependent on the presence of a CORBA ORB. If an ORB is not present, `CommandBrokerImpl` simply cannot be invoked by means of polymorphism through the `POA_CommandBroker` skeleton.

The HTTP, DCE, and CORBA components of this architecture can be removed without alteration of the architecture itself.

7.3.2.2 The OODCE/CORBA Server

A similar migration architecture, the OODCE-over-CORBA architecture is presented here for the purpose of comparison.

The OODCE-over-CORBA architecture is similar at a high level, but omits all command brokering because that is managed by the Object Request Broker (ORB).

The OODCE-over-CORBA architecture depends on the same runtime support libraries as the OODCE-over-XML architecture:

DCEServer

Initializes CORBA ORB

DCEPThread

The OODCE-over-CORBA architecture is very similar to the OODCE-over-XML architecture in the way it re-designs OODCE stubs and utilizes data type translators below the OODCE method call level:

```

<iface>_<ver>_Mgr
    is-a <iface>_<ver>_ABS
        is-a oodceBase
            contains ServantBase;

CORBA_<iface>
{
    calls
    {
        <iface>_<ver>_Mgr;
        TypeMapper;
    }

    contains <iface>_<ver>_ABS;

    is-a ServantBase;
}

```

7.3.2.3 The OODCE/CORBA Client

The client OODCE/CORBA architecture is simpler yet:

```

<iface>_<var>
{
    contains
    {
        <iface>_var;
        CosNaming::NamingContext_var;
    }
}

<iface>_var
{
    is-a CORBA::_var;

    contains <iface>
}

DCEPThread

```

The DCEPThread facility is accessible to both clients and servers.

7.3.2.4 The OODCE/XML Client

The client OODCE/XML architecture resembles that of the OODCE/CORBA strategy at the call level, but adds XML marshalling code underneath:

```

<iface>_<var>
{
    is an OODCE_SOAP_Wrapper;

    calls
    {
        TypeMapper;
        SoapCommand;
        SoapResponse;
        XmlParser;
    }
}

OODCE_SOAP_Wrapper
{
    contains
    {
        CommandBroker_var or
            handle_t;
        CosNaming::NamingContext_var;
    }

    calls
    {
        CommandBroker_var (CORBA) or
            CommandBroker (DCE);
    }
}

```

```

XmlParser
{
    cont
    {
        XERCES::SAXParser;
        NonParseHandlers
        is-a XERCES::HandlerBase;
    }
}

SoapCommand
{
    contains { SoapParamBlock cont SoapParam; }

    is-a SoapMessage;
}

SoapResponse is-a SoapMessage;

DCEPThread

```

7.4 Case Specifics

Please see section 5.5 for a description of the specifics of the DCE IDL used in this case. The section discusses two DCE IDL interfaces:

- math (5.5.1)
- subscription (5.5.2)

7.5 Process

This section briefly describes the process of porting example (OODCE) applications using the tools developed for this project, and executing the resulting ported (SOAP) applications.

Two example OODCE IDL interfaces are discussed:

- math
- subscription

The process of porting OODCE applications to SOAP is similar to that of CORBA (see section 5.6), only it doesn't require any use of a CORBA IDL compiler, because the

entire stubs are generated by the OODCE-to-SOAP converter. Because of this, the OODCE-to-SOAP conversion process does not involve CORBA IDLs.

The examples were executed in the Microsoft Windows 2000 environment.

The steps of test execution are as follows:

1. Process each interface (IDL) file with the "Converter" utility, which replaces the *idl++* utility with respect to high-level code generation.
2. Create client and server projects consisting of legacy code and the output of step 1.
3. Build the client and server executables.
4. Launch the server application(s).
5. Launch the client application(s).
6. Record server and client output.

Once the server processes are launched, the client program is launched and its output monitored. The objective is to see that the client binds successfully to all server objects, and succeeds in receiving the proper results from each server object's member functions.

7.5.1 Simple and Threaded Math Examples

The interfaces and implementations (client and server code) used for the math examples were provided for this project

The examples provided for this exercise use two similar interfaces: *math* and *matha*. For the purposes of migration to SOAP, these interfaces are identical. The example is constructed such that a server program may register one of two pairs of objects with the ORB, depending on how that server program is launched. Since the client expects to bind to both object pairs, the server program is launched twice (each launch in a separate process).

7.5.2 Subscription Server Example

The interface used for the subscription example was provided for this project by NASA. Client and server implementation code was not provided by NASA, so several example implementations were developed during the course of project activities.

7.6 A Code Walk Through an OODCE/XML Remote Invocation

For this example code walk, an OODCE/SOAP client that uses HTTP as a transport mechanism is used. The walk begins with the issuance of an invocation of an OODCE method.

7.6.1 The Path of the Function Invocation Through the Client

First, the client invokes a method by way of an OODCE look-alike stub:

```
DsSdDCESubscrHandle mySubscription =  
    SubscrService.CreateNewSubscription(  
        theData,  
        theDataLength );
```

This call invokes the OODCE stub method, defined as followed by OODCE's IDL++ compiler:

```
Class DsSdDCESubscrIDL_1_0
{
    ...

    DsSdDCESubscrHandle CreateNewSubscription(
        Idl_byte data[],
        Idl_long_int len );
```

The OODCE stub implementation is replaced by the following implementation for execution over SOAP:

```
::DsSdDCESubscrHandle
DsSdDCESubscrIDL_1_0::CreateNewSubscription(
    idl_byte data[],
    idl_long_int len )
{
    if (initialized)
    {
        CORBA::ULong out =
            theObject->CreateNewSubscription(
                *to_octetArray(data, len),
                len );

        return (void*)out;
    }
    else
    {
        cerr << "cannot perform operation without
binding.\n";
        return (DsSdDCESubscrHandle) -1;
    }
}
```

For execution of OODCE client calls via SOAP, data are converted to and from XML,
rather than CORBA types:

```

::DsSdDCESubscrHandle
DsSdDCESubscrIDL_1_0::CreateNewSubscription(
    idl_byte data[],
    idl_long_int len )
{
    if (initialized)
    {
        SoapCommand* command =
            new
            SoapCommand("servant.CreateNewSubscription" );

        command->addParam(
            "data",
            TypeMapper::from_idl_byte_array(data, len) );

        command->addParam(
            "len",
            TypeMapper::from_idl_long_int(len) );

        BrokerString responseString =
            issueSoapCommand(command);

        SoapResponse* response =
            parser->getSoapResponse(responseString);

        return TypeMapper::to_void_ptr(
            response->getValue());
    }
    else
    {
        cerr << "cant perform operation without binding.\n";
        return (DsSdDCESubscrHandle) -1;
    }
}

```


Transport details are encapsulated within the `issueSoapCommand` method of the base class `OODCE_SOAP_Wrapper`. This method is what actually issues the HTTP request:

```
BrokerString
OODCE_SOAP_Wrapper::
IssueSoapCommand(SoapCommand* command)
{
    BrokerString messageBuffer = command->toString();
    size_t messageLength =
        XMLString::stringLen(messageBuffer) * 2;

    return (BrokerString) sendHTTPRequest(
        (char*) messageBuffer,
        messageLength );
}
```

The SOAP command that becomes the body of the HTTP request appears as follows:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>

    <m:DsSdDCESubscrIDL.CreateNewSubscription xmlns:m=
      "http://www.kaweah.com/schemas/soap/dce">

      <h xsi:type="long">DIS</h>

      <data xsi:type="SOAP-ENC:base64">
        aG93IG5vDyBicm73biBjb3cNCg==
      </data>

      <len xsi:type="long">19</len>

    </m:GetLastTradePrice>

  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The outer element of the SOAP message is the envelope. Within the envelope is the encoding style element, and then with that element is the message body. The sole element directly within the body is the command `DsSdDCESubscrIDL.CreateNewSubscription`, which contains the parameters `h`, `data`, and `len`, as specified by the interface IDL.

In this implementation, the server manager handle `h` is encoded as an unsigned long integer, though it could alternatively be encoded as a byte array.

The second parameter `data` is be encoded as a byte array. The suggested data transfer encoding for SOAP is Base64.

The final parameter `len` is encoded as a long, as specified by the DCE IDL and the DCE IDL-to-C type mapping. This parameter will specify the length of the `data` array to the server, though at this stage it is not strictly necessary because the `data` array is passed by value.

For this implementation, the `sendHTTPRequest ( )` function constructs an HTTP request using the SOAP command as the request body and issues the request over a socket. HTTP is implemented as discussed in the section entitled “Implementing HTTP for DCE/SOAP”.

7.6.2 The Path of the Function Invocation Through the Server

The invocation appears on the server in the HTTP server code:

```
// Receive data from the client
memset(szBuf, 0, sizeof(szBuf));
nRet = recv(remoteSocket,          // Connected client
            szBuf,                 // Receive buffer
            sizeof(szBuf),        // Length of buffer
            0);                   // Flags

if (nRet == INVALID_SOCKET)
{
    PRINTERROR("recv()");
    closesocket(listenSocket);
    closesocket(remoteSocket);
    return -6;
}

CommandBrokerImpl *theBroker = theServer->getBroker();
BrokerString response = theBroker->issueCommand((BrokerString)szBuf);
```

Though the signature of this function includes the server handle parameter "h", the server that is identified by the handle will be the server receiving the call at runtime, so there is little or no need to use the handle, nor will any such DCE-specific handles be utilized by the CommandBroker.

The `CommandBrokerHandle c` identifies the `CommandBroker` to be summoned in issuing the command. Multiple `CommandBrokers` may operate concurrently on a single server manager.

The SOAP command to be executed is contained in the `BrokerString` command parameter.

In this example, the DCE implementation is calling the CORBA interface implementation class `CommandBrokerImpl`. The specific method, `issueCommand()`, performs a sequence of operations on the command:

```
char*
CommandBrokerImpl::
issueCommand(const char* command)
{
    MemBufInputSource* commandBuffer
        = new MemBufInputSource(
            (const XMLByte*) command,
            strlen(command),
            gMemBufId,
            false );

    int rc = parser->parse(commandBuffer);

    if (rc != 0)
    {
        sprintf(resp, "Parser failed with code %i\n", rc);
        return resp;
    }

    SoapCommand* theCommand = parser->getSoapCommand();

    ByteString name =
        SoapMessage::wide2byte(theCommand->getName());

    ByteStringTokenizer* tokens =
        new ByteStringTokenizer(name);

    (void) tokens->nextToken(":"); // get past "m:"
    ByteString* objectName = tokens->nextToken(".");
    ByteString* method = tokens->nextToken();

    oodceBase* object = getServant(*objectName);

    SoapResponse* response;
```

```

if (object != NULL)
{
    response = object->execute(
        *method,
        theCommand->getParams() );
}
else
{
    response = new SoapResponse(
        *objectName,
        L"Object not found." );
}

```

The command is parsed, converted into a SoapCommand object, then passed to the

oodceBase::execute() pure virtual function:

```

virtual SoapResponse*
execute(
    ByteString      methodName,
    SoapParamBlock* params
) = 0;

```

`execute()` is implemented by the `<interface>_<version>_ABS` abstract class. In this implementation, the call parameters are extracted from the `SoapCommand` and passed to the appropriate interface method:

```
SoapResponse*
DsSdDCESubscrIDL_1_0_ABS::
execute(    ByteString      methodName,
          SoapParamBlock*   params )
{
    XMLCh* returnString;

    DsSdDCESubscrIDL_1_0_XML* impl =
        new DsSdDCESubscrIDL_1_0_XML(NullUuid, this);

    if (methodName == "CreateNewSubscription")
    {
        returnString = impl->CreateNewSubscription(params);
    }
}
```

The `ParamBlock` is passed to the selected `xml_<iface>_<ver>_XML` method, which extracts each method parameter and converts it from unicode to a binary format expected by the OODCE server implementation.

```

XMLCh*
DsSdDCESubscrIDL_1_0_XML::
CreateNewSubscription(SoapParamBlock *params)
{
    XMLCh* dataStr = new XMLCh[ 256] ;
    XMLCh* lenStr = new XMLCh[ 256] ;

    XMLString::copyString(dataStr, params->getValue("data"));
    XMLString::copyString(lenStr, params->getValue("len"));

    idl_long_int len = TypeMapper::to_idl_long_int(lenStr);

    TypeMapper::idl_byte_array* data =
        TypeMapper::to_idl_byte_array(dataStr);

    DsSdDCESubscrHandle out =
        oodceLayer->CreateNewSubscription(
            data->getData(),
            len );

    return TypeMapper::from_void_ptr(out);
}

```

The conversion of each parameter consists of passing that parameter to a TypeMapper function. For example:

```

idl_byte*
TypeMapper::
to_idl_byte(XMLCh* str)
{
    unsigned int len = XMLString::stringLen(str);

    idl_byte *array = new idl_byte[ len] ;

    for (unsigned int i = 0; i < len; i++)
    {
        array[i] = (idl_byte) str[i];
    }

    return array;
}

```

The translated data is then passed to the OODCE function:

```
out = oodceLayer->CreateNewSubscription(data, len);
```

This pure virtual function is a member of the `DsSdDCESubscrIDL_1_0_ABS` abstract class, which is subclassed by the `DsSdDCESubscrIDL_1_0_Mgr` concrete class, as determined by OODCE. The implementation of the `DsSdDCESubscrIDL_1_0_Mgr` method might look as follows in a simplified case:

```
DsSdDCESubscrHandle
DsSdDCESubscrIDL_1_0_Mgr::
CreateNewSubscription(
    idl_byte data[],
    idl_long_int len)
{
    subscription* theSubscription =
        new subscription(data, len);

    subscriptionArray[ subCount] = theSubscription;

    subCount++;

    return (DsSdDCESubscrHandle) theSubscription;
}
```


7.6.3 The Path of the Function Return through the Server

The OODCE server implementation returns a data value to the `xml_<iface>` method, which passes that value to a translator before passing it on to the command broker layer:

```
out = oodceLayer->CreateNewSubscription(
    data,
    len );

return TypeMapper::from_void_ptr(out);
```

In the above example, the translator would resemble the following:

```
XMLCh*
TypeMapper::
from_void_ptr(void* in)
{
    XMLCh *out = new XMLCh[SOAP_PARAM_MAX_CHARS +1];

    XMLString::binToText(
        (long) in,
        out,
        SOAP_PARAM_MAX_CHARS,
        10 );

    return out;
}
```

The SOAP-compatible data is then returned to the `oodceBase::execute()` method:

```
returnString =
    impl->CreateNewSubscription(theCommand->getParams());
```

The `execute()` method then wraps the return value string in a SOAP response object and passes that response back to the `CommandBroker`:

```
return new SoapResponse(methodName.toWord(), returnString);
```

The `CommandBroker::issueCommand` method then passes the stringified response back to the transport layer:

```
if (object != NULL)
{
    response = object->execute(
        *method,
        theCommand->getParams() );
}
else
{
    response = new SoapResponse(
        *objectName, L"Object not found.");
}

return SoapMessage::wide2byte(response->getResponse());
```

Finally, the response string is given an HTTP response header and delivered to the client by way of a socket:

```
BrokerString response =
    theBroker->issueCommand((BrokerString)szBuf);

char* HTTPResponseBuffer = new char[1024];
size_t responseLength;

char* HTTPResponse =
    DCEServer::constructHTTPResponse(
        response,
        HTTPResponseBuffer,
        &responseLength );

// Send data back to the client

nRet = send(
    remoteSocket,
    HTTPResponse,
    responseLength,
    0 );          // Flags

closesocket(remoteSocket);
```

Alternatively, the transport mechanism could be CORBA or DCE. If it is a CORBA ORB, the response string would be passed back through the POA_<i face> abstract base class:

```
class POA_CommandBroker :
public      CommandBroker_ops ,
public      virtual PortableServer_ServantBase
{
    ...

    // The following operations need to be implemented

    virtual char* issueCommand(const char* _s) = 0;
```

If the transport mechanism is DCE, the response string is passed back to the DCE

CommandBroker function implementation:

```
unsigned char __RPC_FAR *issueCommand(  
    /* [in] */ handle_t h,  
    /* [in] */ BrokerString command)  
{  
    CommandBrokerImpl *theBroker = theServer->getBroker();  
    return (BrokerString)  
        theBroker->issueCommand((const char*)command);  
}
```

7.6.4 The Path of the Function Return through the Client

The command response is passed back from the `sendHTTPRequest()` function (or alternatively a CORBA stub or DCE client stub) into the OODCE client function `DsSdDCESubscrIDL_1_0::CreateNewSubscription()`. The return message is then converted to the data type appropriate to the interface definition, then passed back to the client application.

```
BrokerString responseString =
    issueSoapCommand(command);

SoapResponse* response =
    parser->getSoapResponse(responseString);

return TypeMapper::to_void_ptr(response->getValue());
```

Finally, the return value is stored in the client, as `mySubscription` in this case:

```
DsSdDCESubscrHandle mySubscription =
    SubscrService.CreateNewSubscription(
        theData,
        theDataLength );
```

7.7 Implementing HTTP for DCE-over-SOAP

SOAP-over-HTTP applications often use popular client and server products such as Microsoft Internet Explorer (IE) and Internet Information Server (IIS), Netscape Navigator, and the iPlanet and Apache web servers, but these products do not always provide the most efficient or manageable SOAP-over-HTTP strategy.

7.7.1 The Multiple Process Dilemma

In the case of DCE-over-SOAP, or any architecture that is linked statically or dynamically to legacy application code, the fact that the client and server legacy code are applications requires that if the HTTP facility is an application, it must be joined to the legacy application by way of inter-process communication (IPC).

The following figure illustrates connectivity options between legacy OODCE applications and popular web client and server products:

Figure 13: Inter-Process Communication

Server connectivity options are restricted further by the fact that OODCE applications are always coded in C++. This means that Java-specific technologies such as browser applets and Java servlets must have some means of communicating with a C++ program at runtime. For this reason, servlets have been omitted from the server connectivity figure above.

For this project, the following modules were developed:

- ISAPI Server Plug-in
- NSAPI Server Plug-in
- Windows IPC connectors

7.7.2 Embedded HTTP facilities

The most direct, simple, and efficient approach is to avoid IPC entirely by embedding dedicated HTTP facilities within the migration layer of the DCEServer and OODCE_SOAP_Wrapper (client side) class.

The following diagram displays some options for embedding HTTP facilities within the migration layer:

Figure 14: Internal HTTP facilities

Heavily shaded boxes in the above diagram indicate the specific options selected for this project. A separate decision must be made for client and server sides of the application. For this project, an in-house, native sockets-based HTTP library strategy was adopted for both client and server sides of the application. The specific native sockets API that was selected was that of Microsoft Windows (WinSock).

Other options were evaluated, for instance, (1) W3C's "libwww" web client library, and (2) the "Simple Sockets" library developed at NASA.

7.7.3 Bridging

It should be noted that a SOAP/HTTP-to-DCE bridge can be implemented using a web server as a foundation. This would enable SOAP/HTTP clients to communicate with legacy DCE applications without any need for any special IPC. This would not be a full SOAP solution, however, as SOAP would be used only on the client side.

7.8 Observations

XML parsing and data translation from OODCE to SOAP and back is an involved task. On the other hand, both the extensibility of XML and the advanced assortment of predefined data types that XML provides make SOAP appear to be a more robust target for translation of DCE data types. The most obvious reason for this is that SOAP and XML are much better documented than DCE-RPC and IIOP, and are therefore more open and accessible. Beyond this, XML supports types such as varying arrays more seamlessly than CORBA, and the language and logic in the XML specification is more akin to that of DCE than CORBA.

SOAP suffers from a number of limitations: [9.5]

- SOAP does not offer any of the high-level services provided by DCE, OODCE, COM, COM+, or CORBA.
- SOAP call packaging is not as efficient as that of COM or IIOP over the wire.
- SOAP does not provide any specification for bi-directional communication (though such communication is easily facilitated with XML).
- SOAP does not address object activation issues.

- SOAP does not address object lifecycle management issues.

Note that some of these limitations are functional (including performance issues), and others are more a matter of standardization. The fact that SOAP does not provide a specification for something does not impact directly on functionality.

SOAP is too limited in scope to be seen as a replacement for DCE, COM, or CORBA, but it may be argued that in some situations SOAP could be used instead of lower-level subsets of these more extensive technologies such as DCE-RPC and IIOP. SOAP is in fact functionally similar to IIOP,⁵⁰ though it is structurally dissimilar.

These limitations may not prove prohibitive for developers who are looking for a simpler, more lightweight facility than those offered by OSF/OG, OMG, or Microsoft.

On the other hand, the manager of the large legacy application may not see much value in SOAP's simplicity or its close relationship to HTTP.

⁵⁰ <http://www.develop.com/soap/soapfaq.htm#14>

SOAP is not yet a standard, but rather is an evolving specification, and provides no established implementations that enterprise applications can rely upon. Still, at a conceptual level, SOAP and XML appear to provide an elegant solution for many developers of Internet client/server applications.

7.9 Outcome

Example frameworks have been developed in support of this thesis for operating OODCE applications by way of SOAP messages, and used HTTP, DCE-RPC and IIOP as command and response transport mechanisms.

This simple OODCE application operates over HTTP, CORBA, and DCE while concealing the details of these transport mechanisms. It demonstrates the potential usefulness of abstracting object invocations by way of XML.

8 Building Bridges from Adapters

The concept of an adapter that converts incoming SOAP or CORBA messages to local invocations of a DCE interface implementation can be easily extended to a bridge by replacing that same local DCE interface implementation with DCE RPCs. Each method implementation, rather than being implemented with application code, is instead implemented with simple calls to a remote DCE implementation.

8.1 A SOAP Bridge

A demonstration bridge has been developed as part of this investigation that is based on the same adapter code used for local DCE-to-SOAP porting described above. The needed changes are as follows:

Instead of using a legacy server implementation, create a pseudo-implementation that simply acts as a DCE client, calling a remote DCE server implementation that is identified when the invoked class is loaded by the bridge application.

Modify the manager (`_MGR`) class definition to include a DCE target handle that refers to the remote DCE server implementation.

Modify one or more manager (`_MGR`) class constructors to receive the newly-introduced DCE target handle as an argument.

Modify the IDL compiler to generate the code changes described in 1-3.

In the following diagram, the client and server, both remote to the bridge, are represented by bold-line boxes. Auto-generated code is represented by dotted-line boxes. The progress of the method invocation from client to server is portrayed in nine steps.

Figure 15: DCE Method Invocation via the SOAP-DCE Bridge

The flow and translation of procedure calls is straightforward, though rather unintuitive in implementation. Server management is less direct a concept. For this implementation, the bridge is treated like a DCE server application that passes the name info (or UUID) of the true server object to each bridge server object. This means the bridge `main()` procedure must be recompiled and the bridge re-linked to add support for new interfaces and servers.

This primitive design could easily be replaced by a program that instantiates objects dynamically according to demand. It would still be a *static* bridge.

By virtue of the necessity of developing a brokering layer for the OODCE/SOAP solution, this bridge includes the same brokering facility. This provides an easy interface to SOAP clients, but is not a necessity in this model, because SOAP clients can provide naming and interface information to the bridge that the bridge could in turn use to find the true DCE server objects, but this would require significant recoding of the bridge's brokering facility.

8.2 A CORBA-DCE Bridge Architecture

The same adapter-based strategy can be applied to developing a CORBA-DCE bridge, however, because a commercial off-the-shelf (COTS) CORBA-DCE bridge is already available, the development of such a bridge has not been attempted as part of this investigation, though the code changes it would require would be very similar to those of the SOAP-DCE bridge.

9 Summary of Development Activities

The activities supporting this thesis began with an effort to re-implement OODCE stubs and runtime libraries to operate over CORBA. This was first achieved for the simple math interface and threaded math interface supplied by NASA.

The next step was to develop a conversion utility to generalize the solution for all DCE interfaces. This was achieved, but many DCE IDL elements remained unsupported.

The next activity in this investigation was an effort to extend the solution to a more typical, more complex interface. The ECS Subscription interface was selected as a good example. Once the OODCE-to-CORBA migration framework fully supported the Subscription interface, the conversion utility was extended to generalize the new, more feature-complete framework.

After completing the migration framework and conversion utility for migration to CORBA, the solution was modified and extended to support migration to SOAP. Because SOAP does not include a transport mechanism, DCE and CORBA were used for transport, though HTTP would have been a more typical choice.

Besides these development efforts, exploratory efforts were directed at XML-RPC, ONC (Sun) RPC, and DCOM.

Example applications were developed and executed for performance testing of DCE, OODCE, CORBA, SOAP, and bridging technologies.

10 Performance

10.1 Summary

Performance tests have shown significant performance degradation resulting from the translation process, though the tested code is *not* optimized, so there is room for improvement.

The OODCE-over-IIOP solution ran about 25% slower than a similar CORBA application. This figure represents the approximate overhead of the OODCE-over-IIOP migration architecture.

Three transport platforms have been selected to support the OODCE-over-SOAP migration mechanism: HTTP, NT/DCE, and VisiBroker. DCE provides the best performance, and CORBA is not very far behind (32% slower). The HTTP implementation developed for this study is about seven times slower than the SOAP/DCE mechanism.

The significant reduction in performance when using HTTP as a transport is probably caused by the overhead incurred while reconstructing sockets for each HTTP connection. Though HTTP is a sessionless protocol, performance improvements may

yet be attained by sharing sockets between connections, but to this point such optimizations have not been implemented and may be regarded beyond the scope of this investigation.

The following table summarizes the time expended for remote procedure calls (RPCs), relative to a local SOAP-over-DCE RPC:

Table 20: Relative Latency of SOAP Transports

Connection Type	DCE	CORBA	HTTP
Local	1.00	1.34	15.84
Network	2.41	3.15	17.73

CORBA is about 32% slower than DCE as a transport for OODCE-over-SOAP method invocations over a network. HTTP is less efficient than DCE and CORBA by a large margin: a factor of 6 to 7.

10.2 Details

The following two tables show the test machines used (Table 21) and the results obtained (Table 22) from testing on those machines.

Table 21: Test Machines

Name	Microprocessor	Clock Speed (MHz)
A	Intel Pentium® III	500
B	Intel Pentium® III	133
C	Intel Celeron® (f6m6s10)	400
D	Intel Pentium® III (f6m8s3)	931

The four test machines were not used for the same test set, nor were they used at the same time. Systems A and B were used for early test rounds, then systems C and D were used after A and B were no longer available. System D was not used for CORBA testing.

Table 22: OODCE-over-SOAP Performance

Transport Mechanism	Local or Network	Connection (C->S) ⁵¹	Mean Elapsed Time ⁵²
CORBA	Local	A->A	916 ms
CORBA	Local	B->B	2008 ms
CORBA	Local	C->C	1515 ms
CORBA	Network	B->A	3567 ms
DCE	Local	A->A	714 ms
DCE	Local	B->B	1552 ms
DCE	Local	C->C	1057 ms
DCE	Local	D->D	390 ms
DCE	Network	B->A	2728 ms
DCE	Network	D->C	1740 ms
HTTP	Local	A->A	10590 ms
HTTP	Local	B->B	24170 ms
HTTP	Local	C->C	17880 ms
HTTP	Local	D->D	8965 ms
HTTP	Network	B->A	18570 ms
HTTP	Network	D->C	14340 ms

The test results for CORBA and DCE based modules should be compared to previous research [6.2].

⁵¹ Client->Server.

⁵² Average elapsed time for a test program consisting of 100 iterations of a 10-call loop (1000 RPCs total).

10.3 OODCE-over-SOAP Overhead

The following table shows a significant performance loss between two transports (DCE and CORBA) and the migration architectures developed for this study, when running over those transports. The loss is substantially greater for the OODCE-over-SOAP architecture:

Table 23: Overhead

Client	Server	Calls per ⁵³ second
DCE	DCE	5700
CORBA	CORBA	4800
DCE/CORBA	DCE/CORBA	3600
DCE/SOAP [trans=DCE]	DCE/SOAP	1300
DCE/SOAP [trans=CORBA]	DCE/SOAP	1150

⁵³ Local invocations on test system A.

Much of the performance degradation associated with the OODCE/SOAP solution can be attributed to the large amount of non-optimized memory management, as every step involves allocation of temporary data that must persist beyond the scope of the allocating routine. The overhead of the migration architecture is 73% (percentage performance loss).

10.4 Bridges

The SOAP-DCE bridge that was developed to support this thesis ran at about 1000 procedure calls per second when using DCE as a transport mechanism⁵⁴. This means the additional performance loss associated with extending the migration architecture to a bridge is about 23%.

⁵⁴ Local invocations on test system A.

Table 24: Bridge Performance

Client	Server	Calls per second ⁵⁵
DCE	DCE	5700
CORBA	CORBA	4800
DCE/CORBA	DCE/CORBA	3600
DCE/SOAP [trans=DCE]	DCE/SOAP	1300
DCE/SOAP [trans=CORBA]	DCE/SOAP	1150
DCE/SOAP [trans=DCE]	DCE [via bridge]	1050
DCE/SOAP [trans=CORBA]	DCE [via bridge]	850

Though these figures reveal significant performance degradation, they compare favorably to similar technologies. Previous research [6.2] has revealed that a CORBA client invoking a DCE server method over a commercial DCE-CORBA bridge is slower by a factor of 11.2 than a DCE client invoking a DCE server method. The DCE-SOAP bridge that was developed for this project compares well at a factor of 5.4. Of course this bridge may be slowed significantly as it is made more mature, but then it could also be improved by performance optimizations.

⁵⁵ Local invocations on test system A.

11 Conclusions

In the general case, developing a migration strategy from a DCE legacy application to another middleware technology such as CORBA or SOAP is a difficult task because some data types, features, and services provided by DCE are not provided by CORBA or SOAP.

DCOM is a convenient migration target because it is based on DCE RPC, but migration to DCOM may require an operating system transplant from UNIX to Microsoft Windows. This investigation directs most of its attention on CORBA and SOAP because NASA's EOSDIS Core System (ECS) is considered too tightly tied to UNIX. This critical dependency is itself open to investigation, however, limited access to the implementation details of ECS prevents such an investigation at this time.

Any choice between migration targets, whether CORBA, DCOM, or SOAP, depends on the needs of the application. Fortunately, many DCE applications do not depend on the entire DCE feature set. ECS is an example of such an application, though it does depend on several non-portable elements of DCE.

CORBA provides command brokering, routing, efficient command and response packaging (IIOP), and transport. CORBA also provides a powerful suite of services, and promises more in the future. Though CORBA has lost much of its

services, and promises more in the future. Though CORBA has lost much of its momentum in the marketplace, it continues to sustain its dominant position in the distributed systems marketplace.

SOAP offers a simple, accessible means of command and response packaging that is tightly coupled with XML and HTTP, yet can use a variety of transport mechanisms. SOAP offers very little in the way of mature products and services, but promises a future of truly open development. By the same token, future developments in SOAP may diverge from any *early adopter* SOAP applications developed in this early stage of SOAP's evolution.

Whereas CORBA threatens to collapse under its own weight in some markets, SOAP is refreshingly lightweight and adaptable. With respect to developing tools for migration, SOAP allows the developer to manage data types at a low level without the learning curve required from the more complex IIOP specification. CORBA stubs and skeletons may be used to avoid the complexities of IIOP, but only at the price of reduced portability and flexibility.

At present, the SOAP specification is incomplete and uncertainties abound regarding the form that SOAP will take once mature. For example, the SOAP specification does not address security at all. This practical consideration, and other

considerations presented above, compel this thesis to arrive at the qualified conclusion that CORBA is the preferred target for migration.

12 Development Tools and Environment

Various tools and environments were used for the purpose of reviewing the technologies discussed in this paper. The DCE/SOAP and DCE/CORBA projects described herein were developed and tested using the following:

- Microsoft Windows 2000 Server
- Borland VisiBroker for C++, v3.3-4.1
- Microsoft Visual Studio v6.0
- Microsoft Visual C++ v6.0
- Apache Xerces-C++ v.1.4.0

13 CORBA 2.x ORBs with C++ Bindings

13.1 Commercial ORBs

Rows with gray backgrounds represent previous releases.

Table 25: Commercial C++ ORBS

Developer	Product	URL	Platform
Borland	VisiBroker for C++ 4.0	http://www.borland.com/techpubs/visibroker/platforms/vbcpp40.html	Solaris 2.6/7/8, HP-UX 10.2/11, AIX 4.2/4.3, Red Hat Linux 6.1, NT, Win 95/98, Win NT 4.0/ 2000, SGI IRIX 6.5
Borland	VisiBroker for C++ 4.1	http://www.borland.com/techpubs/visibroker/platforms/vbcpp41.html	Same as v4.0, less Win95 and SGI IRIX 6.5. ⁵⁶
Borland	VisiBroker for C++ 4.5	http://www.borland.com/techpubs/visibroker/platforms/vbcpp45.html	Same as v4.1, less AIX 4.2 and HP-UX 10.2; Red Hat Linux 7.0 replaces 6.1.
BEA	WebLogic Enterprise	http://www.bea.com/products/weblogic/enterprise/index.shtml	HP-UX 11, Solaris 2.6/7, NT 4.0/2000, AIX 4.3, Compaq Tru64 UNIX 4.0F, OS/390 Unix System Services, SCO UnixWare 7.1.1
Sun	NetDynamics 4	www.netdynamics.com	NT 4.0, Solaris 2.6, AIX 4.2/4.3, HP-UX 10.2
Sun	NetDynamics 5	www.netdynamics.com	NT 4.0, Solaris 2.6 ⁵⁷ , AIX 4.3, HP-UX 11
Iona	Orbix 3.3 (C++) [CORBA 2.1]	http://www.iona.com/products/orbix3_home.htm	Solaris 7/8, HP-UX 11, NT 4.0, AIX 4.3, Compaq Tru64

⁵⁶ No current commercial CORBA/C++ products support SGI IRIX. Borland's *VisiBroker* and BEA's *WebLogic Enterprise* were the last.

⁵⁷ FAQ (<http://www.sun.com/products/netdynamics/products/faq.html#dist>) claims support for Solaris 7, but the product release notes (<http://docs.ipplanet.com/docs/manuals/nd/relnotes.html>) do not.

Iona	Orbix 2000 (C++) [CORBA 2.4]	http://www.iona.com/products/orbix2000_home.htm	UNIX 5.1 Solaris 2.6/8, HP-UX 11/11i, Win98, NT 4.0/2000, AIX 4.3, Compaq Tru64 UNIX 5.1, Red Hat Linux 6.2
IBM	WebSphere Application Server, Enterprise Edition 3.6	http://www-4.ibm.com/software/web servers/apps erv/enterprise.html	AIX 4.3, Solaris 7/8, NT 4.0/2000
Secant	ModelMethods™ Enterprise Server [CORBA 2.3]	http://www.secant.com/products/ES/	NT 4.0/2000, Win95/98 (client only), Solaris 2.6/7/8, HP-UX 11, Linux, BSD/OS
Expersoft	eORB for C++ [CORBA 2.2 "minimumCORBA"]	http://www.expersoft.com/Products/CORBAC/corbac.htm	LINUX, Solaris (SPARC and x86), AIX, HP-UX, VxWorks ® , etc.

13.2 Non-Commercial ORBs

Table 26: Non-Commercial C++ ORBs

Developer	ORB	URL	Version	Platforms
Iona/OOC	ORBacus 4	http://www.orbacus.com/	2.3	Sources for NT, UNIX
Distributed Object Computing G., WUSTL	TAO	http://www.cs.wustl.edu/~schmidt/TAO.html	~2.4	Sources ported to Win95/98/CE, NT 3.5/4.0/2000, Solaris, SGI IRIX 5-6, HP-UX 9-11, AIX 3-4, ...
Bionic Buffalo Corp.	Tatanka ORB ⁵⁸	http://www.tatanka.com/prod/info/orb1.htm	2.2	Linux, Solaris, FreeBSD, or Windows
AT&T	OmniORB 3.0	http://www.uk.research.att.com/omniORB/omniORB.html	2	Sources supported for Solaris 2.5/2.6, Win-95/NT, Linux 2.x, HP-UX 11, SGI IRIX 6.x

⁵⁸ C language mappings supported; C++ mappings under development.

14 Works Cited

14.1 RPC

- [1.1] Power Programming with RPC. John Bloomer, O'Reilly & Associates, Inc., 1992.
- [1.2] Guide to Writing DCE Applications. John Shirley, Wei Hu, and David Magid. O'Reilly and Associates, Inc. Second Edition; Second Printing. November 1994.
- [1.3] RPC Protocol Specification. Version 2. <http://www.freesoft.org/CIE/RFC/1831>
- [1.4] OSF DCE Questions and Answers -- Remote Procedure Call.
<http://www.fi.infn.it/DFS/FAQ/qna/faq02.html>
- [1.5] Remote Procedure Calls: The Next Best Thing to Being There. Sara J. Gilbertson.
<http://hal.csd.auth.gr/unix-books/rpc1>
- [1.6] Building Secure and Reliable Network Applications. Kenneth B. Birman
<http://www.cs.cornell.edu/Info/Courses/Spring-97/CS514/book/bhtml.html>

14.2 DCE

- [2.1] Understanding DCE. Ward Rosenberry, David Kenney, and Gerry Fisher. O'Reilly and Associates, Inc. First Edition; Second Printing. May 1993.
- [2.2] Open Group Technical Standard: DCE 1.1 Remote Procedure Call. The Open Group, Document C706. August 1997. <http://www.opengroup.org/public/pubs/catalog/c706.htm>
HTML: <http://www.opengroup.org/onlinepubs/009629399/>
- [2.3] DCE Frequently Asked Questions. Open Group. 23 October 1998 (still current on 22 June 2001).
<http://www.opengroup.org/tech/dce/info/faq-mauney.html>
- [2.4] DCE 3.1 for Solaris. IBM Pittsburgh Lab.
<http://www.transarc.ibm.com/Support/dce/dce31.html>
- [2.5] DCE 1.1 Online Documentation. Large Integrated Online Networked Services, Old Dominion University. <http://www.lions.odu.edu/docs/dce/>
- [2.6] DCE Today. David Chappell, Open Group.
http://www.opennc.org/dce/info/dce_objects.htm
- [2.7] OSF DCE Application Development Reference. Transarc. Revision 1.1.

14.3 OODCE

[3.1] HP Object Oriented DCE C++ Class Library Programmer's Guide. November 1994. E1194; Second Edition.

14.4 DCE and CORBA

[4.1] Software Tools for Automating the Migration from DCE to CORBA. Douglas C. Schmidt, Aniruddha Gokhale, and Stanley Moyer. In *Proceedings of ISS 97: World Telecommunications Congress*, IEEE Toronto, Canada, September, 1997.
<http://www.cs.wustl.edu/~schmidt/PDF/ISS-97.pdf>

[4.2] Strategies for Migrating from DCE to CORBA. Douglas C. Schmidt and Aniruddha Gokhale.

[4.3] Douglas Schmidt's PDF archive at Washington University
<http://www.cs.wustl.edu/~schmidt/PDF/>

[4.4] Middleware for Building Distributed Application Infrastructure. Fatoohi, R., McNab, D., and Tweten, D., NAS Technical Report NAS-97-026, NASA Ames Research Center, Moffett Field, California, 1997.

14.5 CORBA

[5.1] Advanced CORBA Programming with C++. Michi Henning and Steve Vinoski. Addison-Wesley Professional Computing Series. First Edition. February 1999.

[5.2] IOP Complete. William A. Ruh, Thomas Herron, and Paul Klinker. Addison Wesley. First Edition. October 1999.

[5.3] CORBA 2.4.2 Specification. Object Management Group (OMG), February 2001.
<http://www.omg.org/technology/documents/formal/corbaiop.htm>

[5.4] What's Coming in CORBA 3. Siegel, J., Object Management Group, April 2, 2001.
<http://www.omg.org/technology/corba/corba3releaseinfo.htm>

[5.5] SNEAKING UP ON CORBA: The Race for the Ideal Distributed Object Model. Richard Hoffman <rhoffman@nwc.com>. Network Computing , 05/03/99, Vol. 10 Issue 9.
http://mba1.bus.olemiss.edu/bus620/frankie_gulledge/corba/more_corba_articles.htm

[5.6] Client/Server Programming with Java and CORBA. Robert Orfali and Dan Harkey. John Wiley & Sons, Inc., Second Edition, 1998.

[5.7] Instant CORBA. Robert Orfali, Dan Harkey, and Jeri Edwards. John Wiley & Sons, Inc., 1997.

14.6 Bridging

[6.1] Inprise DCE-CORBA Bridge. Borland International (Inprise Corporation).

Home page: <http://www.inprise.com/entera/dcecorba/>

White paper: <http://www.inprise.com/entera/papers/dce-corba/>

[6.2] Performance Evaluation of Middleware Bridging Technologies. Rod Fatoohi, Vandana Gunwani, Qi Wang, and Charlton Zheng. 2000 IEEE Int. Symp. on Performance Analysis of Systems & Software (ISPASS2000) (April 2000, Austin, TX), pp. 34 - 39.

<http://www.engr.sjsu.edu/fatoohi/Bridge.pdf>

[6.3] DSTC On-Demand DCE-CORBA Bridge. Pre-release announcement, October 1996.

<http://archive.dstc.edu.au/BDU/APAP/Bridge/Bridge.html>

[6.4] Understanding any IDL - Lesson one: DCE and CORBA. Andreas Vogel, Brett Gray, and

Keith Duddy. <http://archive.dstc.edu.au/AU/staff/andreas-vogel/papers/sdne96/paper.html>

[6.5] Achieving Interoperability between CORBA and DCE Applications Using Bridges.

Zhonghua Yang and Andreas Vogel. CRC for Distributed Systems Technology. 1996.

<http://archive.dstc.edu.au/AU/projects/iwt/icdp-submission/icdp-submission.html>

[6.6] David Arnold, Andy Bond, and Alexei Vovenko. A Dynamic Invocation Interface for

DCE. In the Proc. of DSTC Symposium, Brisbane, July 1995. DSTC Pty Ltd.

[6.7] OMG Interop Mailing List. Michi Henning, 3 November 1998.

[6.8] Michi Henning: <http://www.triodia.com/staff/michi-henning.html>

14.7 EOSDIS Core System (ECS)

[7.1] Code Walkthrough. Frank DeLuca, ECS Developers Workshop, 30 May 1995.

[7.2] Distributed Objects. Naveen Hota, ECS Developers Workshop, 30 May 1995.

14.8 DCOM

[8.1] COM and CORBA Side by Side. Jason Pritchard, PhD, Addison Wesley, 1999.

[8.2] Learning DCOM. Thuan L. Thai, O'Reilly & Associates, 1999

[8.3] Understanding the DCOM Wire Protocol by Analyzing Network Data Packets. Guy Eddon and Henry Eddon. Microsoft Systems Journal, March 1998.

[8.4] Using the MIDL Compiler, MIDL Command Line Reference.
http://msdn.microsoft.com/library/psdk/midl/mi-cmdl_n_4tk6.htm

[8.5] Microsoft COM Technologies: DCOM. Microsoft Corporation. March 1998.
<http://www.microsoft.com/com/tech/dcom.asp>

[8.6] DCOM - A Technical and Business Overview. Greg Mountford gregm@ttginc.com. TTG, Inc. <http://www.ttginc.com/dcomarticle.htm>

14.8.1 Windows NT Development

[8.7] Archives of the NT Developers mailing list ("ntdev").
<http://www.ntdev.org/archive>

[8.8] DCE on NT. William A. Estrem, 3M Information Architecture Department.
<http://www.ntdev.org/archive/ntdev9509/rmsg0073.html>

14.8.2 DCE Interoperability and Portability

[8.9] Distributing Applications Across DCE and Windows NT. Ward Rosenberry and Jim Teague. O'Reilly & Associates, Inc. 1st Edition. November 1993.

[8.10] Digital UNIX and Windows NT Interoperability Guide. Digital Equipment Corporation. EC-G5855-43. May 1996.
<http://wint.decsy.ru/du/DIGITAL/UNIXNT/CH9.HTM>

14.8.3 Ports of COM/DCOM to UNIX

[8.11] COM for Tru64 UNIX: Product Definition. Compaq Computer Corporation.
<http://www.tru64unix.compaq.com/com/comspd.htm>

[8.12] COM on Compaq Tru64 UNIX and Windows NT/Tru64 UNIX Interoperability. Compaq Computer Corporation.
http://www.tru64unix.compaq.com/com/white/com_unix.htm

[8.13] Building COM Components on UNIX. Christian Gross, euSoft. July 1998.
http://msdn.microsoft.com/library/techart/msdn_unixcom.htm

[8.14] Web + Object Integration. Craig Thompson and Gil Hansen. 16 August, 2000.
<http://www.objs.com/survey/web-object-integration.htm#OLE-DCOM>

[8.15] EntireX Product Site: Technical Data. Software AG.
<http://www.softwareag.com/entirex/technical/data.htm>

14.9 Simple Object Access Protocol (SOAP)

[9.1] WWW Consortium: The SOAP Specification (version 1.1 at time of writing)
<http://www.w3.org/TR/SOAP/>

[9.2] XML Schema Part 2: Datatypes. WWW Consortium (W3C).
<http://www.w3.org/TR/xmlschema-2/>

[9.3] XML Protocol Working Group Issues List. WWW Consortium (W3C).
<http://www.w3.org/2000/xp/Group/xmlp-issues>

[9.4] Simple Object Access Protocol. DevelopMentor
<http://www.develop.com/soap/>

[9.5] SOAP Frequently Asked Questions. DevelopMentor.
<http://www.develop.com/soap/soapfaq.htm>

[9.6] SOAP Developer Resources. Microsoft.
<http://msdn.microsoft.com/soap/>

[9.7] SOAP Specification (version 1.1 at time of writing). Microsoft.
<http://msdn.microsoft.com/xml/general/soapspec.asp>

[9.8] Schema associated with the "xsi" namespace prefix. WWW Consortium (W3C).
<http://www.w3.org/1999/XMLSchema-instance>

[9.9] Schema associated with the "xsd" namespace prefix. WWW Consortium (W3C).
<http://www.w3.org/1999/XMLSchema>

[9.10] .NET from the Enterprise Perspective: SOAP. Jim Farley, September 2000.
http://java.oreilly.com/news/soap_0900.html

[9.11] Microsoft .NET vs J2EE: How Do They Stack Up. Jim Farley, August 2000.
http://java.oreilly.com/news/farley_0800.html

[9.12] Getting to the Bottom of .Net. Jim Farley and Tim O'Reilly, 25 August 2000.
http://windows.oreilly.com/news/farley_oreilly_0800.html

[9.13] Head to Head: Box on SOAP. Matt Nicholson and Don Box (DevelopMentor).
http://www.dnjonline.com/articles/head2head/iss20_head2head.html

[9.14] Mapping CORBA and SOAP. 31 Oct 2000
<http://nenya.ms.mff.cuni.cz/thegroup/seminars/2000-10-31-kalibera-corba-soap/referat.html>

[9.15] Sun changes tune in support of SOAP protocol

Wylie Wong, 7 June 2000.

<http://technews.netscape.com/news/0-1003-200-2033523.html>

[9.16] J2EE vs. Microsoft.NET: A Comparison of Building XML-based Web Services. Chad Vawter and Ed Roman (The Middleware Company), June 2001

<http://www.sun.com/software/sunone/wp-j2ee/J2EE-vs-DotNET.pdf>

[9.17] Open Net Environment (ONE) Software Architecture. Sun Microsystems.

<http://www.sun.com/software/sunone/wp-arch/>