

542-TP-003-001

Release A Communications and Management Subsystems Developer's Guide for the ECS Project

Technical Paper

Technical Paper—Not intended for
formal review or government approval.

August 1995

Prepared Under Contract NAS5-60000

RESPONSIBLE ENGINEER

<u>H. Naveen /s/</u>	8/11/95
Naveen Hota, Communications Engineer EOSDIS Core System Project	Date

SUBMITTED BY

<u>Parag N. Ambardekar /s/</u>	8/11/95
Parag Ambardekar, Release A Office Manager EOSDIS Core System Project	Date

Hughes Information Technology Corporation
Landover, Maryland

This page intentionally left blank.

Abstract

The Communications Subsystem (CSS) and the Management Subsystem (MSS) provides the overall communications infrastructure, and the management infrastructure to support other subsystems in the Science and Communications Development Office (SCDO) and the Flight Operations Segment (FOS). This document provides the developer's directives and guidelines that developers should follow in design and development of all the subsystems in Release A.

Keywords: CSMS, CSS, Communications, DCE, OODCE, Release A, Time, Directory, Naming, Security, Distributed Objects, LifeCycle, Threads, File Transfer, Bulletin Board, Mail, Event Logging, MSS, Agents, User Profile.

This page intentionally left blank.

Contents

Abstract

1. Introduction

1.1	Scope	1-1
1.2	Document Organization	1-1

2. Related Documents

2.1	Referenced Documents	2-1
2.2	Suggested Readings	2-1

3. Overview

3.1	CSS Introduction	3-1
3.2	CSS Services Release A	3-1
3.3	CSS Provided Services	3-2

4. Release A Design

4.1	Introduction	4-1
4.2	Services Description	4-2
4.2.1	Directory Naming Service	4-2
4.2.2	Security Service	4-12
4.2.3	Message Passing	4-23
4.2.4	Multicast	4-43
4.2.5	Thread Service	4-44
4.2.6	Time Service	4-48
4.2.7	LifeCycle Service (Initialization/Activation)	4-50
4.3	Distributed Object Framework	4-54

4.3.1	Overview	4-54
4.3.2	Context	4-56
4.3.3	Directives and Guidelines	4-56
4.3.4	Sample Application Programmer Interface.....	4-57
4.3.5	Object Model.....	4-59
4.3.6	DOF Frequently Asked Questions	4-60
4.4	Common Facility Services	4-73
4.4.1	Email	4-73
4.4.2	FTP	4-75
4.4.3	Bulletin Board	4-77
4.4.4	Virtual Terminal.....	4-78
4.4.5	Event Logging.....	4-78

5. Distributed Object Framework (OODCE)

5.1	Service Description	5-1
5.2	Why Use Distributed Objects?.....	5-1
5.3	Example	5-2

6. System Management

6.1	User Profile External Interfaces	6-1
6.2	Management Agent Services External Interfaces	6-2

Figures

4.2.1.4-2.	Naming Service - CDS Entry Structure	4-5
4.3.4-1	Client/Server Application Development Process	4-57

Tables

3.2-1.	Characterization of Service Requirements by Release	3-2
4.1-1.	CSS Design Section	4-1
4.2.1.5-1.	Naming Service Object Responsibility Matrix.....	4-12

4.2.2.5-1.	Security Object Responsibility Matrix	4-24
4.2.3-1.	Message Passing Communication Types Defined.....	4-26
4.2.3.1.6-1	Message Passing Object Responsibility Matrix	4-35
4.2.3.2.6-1.	Message Passing Object Responsibility Matrix	4-42
4.2.5.3-1.	Thread Service Object Responsibility Matrix	4-47
4.2.6.3-1.	Time Service Object Responsibility Matrix	4-50
4.2.7.5-1.	LifeCycle Service Object Responsibility Matrix.....	4-54
4.3.5-1.	DOF Object Responsibility Matrix.....	4-60

Abbreviations and Acronyms

This page intentionally left blank.

1. Introduction

1.1 Scope

The Developer's Guide describes for each Communication Subsystem-provided service: an overview, context, directives and guidelines, API samples and the object model descriptions.

1.2 Document Organization

The document is organized to describe the Release A Communications Subsystem developer's guide as follows:

Section 1 provides information regarding the scope and the organization of this document.

Section 2 provides a listing of the related documents, which were used as source information for this document.

Section 3 provides an introduction to CSS, the CSS Services of Release A, and brief descriptions of each of the CSS provided services.

Section 4 contains an overview, context, directives and guidelines, API samples and the object model descriptions for each of the CSS provided services.

The section Abbreviations and Acronyms contains an alphabetized list of the definitions for abbreviations and acronyms used in this volume.

This page intentionally left blank.

2. Related Documents

2.1 Referenced Documents

305-CD-012-001 Release A CSMS Communications Subsystem Design Specification.
HP Object Oriented DCE C++ Class Library Programmer's Guide

2.2 Suggested Readings

HP Object Oriented DCE C++ Class Library External Reference Specification

HP Object Oriented DCE C++ Reference Manual

OSF DCE 1.0.3 Application Development Guide

OSF DCE 1.0.3 Application Development Reference

Understanding DCE by Ward Rosenberry, David Kenney, and Gerry Fisher

Guide to Writing DCE Applications by John Shirley

OSF DCE Guide to Developing Distributed Applications by Harold W. Lockhart, jr.

308-CD-001-004 Software Development Plan for the ECS Project

This page intentionally left blank.

3. Overview

3.1 CSS Introduction

The Communications Subsystem (CSS) provides for the interconnection of users and service providers, transfer of information within the Earth Observing System Data Information System (EOSDIS) Core System (ECS) and between ECS and many EOSDIS components, and management of all ECS communications components. It supports and interacts with the Science Data Processing Segment (SDPS) and the Flight Operations Segment (FOS). This section provides the following: overview of the CSS services for Release A and a design characterization for each CSS service.

3.2 CSS Services Release A

CSS provides infrastructural services, characterized as "middleware," for use in all the subsystems of ECS. This infrastructure consists of communication services that application developers will use in the development of distributed applications, which enables these applications to interact with other applications within and outside of ECS. These services are standards-based and are interoperable (hardware and vendor independent). These services are broadly classified into three categories:

- Common Facilities
- Object Services
- Distributed Object Framework (DOF).

The Common Facilities include legacy communications services required within the ECS infrastructure for file transfer, electronic mail, bulletin board and remote terminal support. The Object Services support all ECS applications with interprocess communication and specialized infrastructural services such as security, directory, time, asynchronous message passing and event logging. The Distributed Object Framework is a collection of a set of core object services, collectively providing object-oriented client server development and interaction amongst applications.

Release A provides ftp, virtual terminal and DCE core services: Directory, Security, Time, RPCs, Mail, Bulletin Board, Event Logger, Message Passing and object oriented DCE services along with some enhancements. These are described in detail in later sections. Release A will use a single DCE cell where all the users, platforms, and services are maintained and belong to.

Table 3.2-1 characterizes the major functionality required of and provided by the CSS within the Release A mission. These are briefly described below and subject to further iteration as the ECS design matures. A complete description of CSS services is provided in Section 4 of this document.

Table 3.2-1. Characterization of Service Requirements by Release

Subsystems/ Services	Major Service Req't	Release A Services
Distributed Object Framework	Interoperability framework for OO client-server development & execution	Execution framework for distributed object implementation
Object Services	Interprocess communication and specialized infrastructural services	Additional services via object interfaces (DCE security, time, CDS directory/ naming logging, message passing, threads)
Common Facilities	Legacy services for file access/ transfer, e-mail, bulletin board and remote terminal support	Additional legacy services with applications & security I/Fs (kftp, ktelnet, e-mail, BBS)

For Release A, additional services are needed from CSS to support the TRMM mission's users and SDPS' mission and user services. These include secure implementations of ftp and telnet services, electronic mail (e-mail) exchange with external systems and bulletin board capabilities. The CSS object-encapsulated services will also require additional functionality in support of SDPS and FOS application event collection and processing, interprocess communications, and thread services. The CSS Distributed Object Framework will be provided to encapsulate all CSS services requiring object-oriented, client-server or peer-peer execution.

3.3 CSS Provided Services

Directory Naming Service

The Directory Naming Service provides a reliable mechanism by which distributed applications can associate information with names. Its primary purpose is to allow clients to locate servers. Its capabilities, however, are general-purpose, and it can be used in any application that needs to make names and their attributes available throughout a network.

CSS will provide implementation of both the DNS and the X.500 by supporting BIND and OSF Global Directory Service and OSF Cell Directory Service (CDS). It also provides application programmers the ability to store, retrieve, list information in the locally supported namespaces. The DNS and X.500 namespaces are used to connect the locally supported CDS namespaces. The functionality provided here will be implemented on top of XDS/XOM interfaces. As such, application programmers can use the above mentioned services (store, retrieve, list) in CDS as well as OSF GDS.

Security Service

The security service provides secure transfer of data on local and wide area networks. It provides mechanisms to verify the identity of users, and to determine whether users are permitted to invoke certain operations (authentication and authorization). Transmission of data is protected through the use of checksums and encryption of data. Authentication is provided by trusted third party (secret key) authentication. Authorization is based on Access Control Lists. The protocol used for authentication is Kerberos. All of these features are implemented within the ECS domain by employing OSF/DCE Security Services.

Multicast

Multicasting is a mechanism through which a single copy of data is transferred from a single point to several places. Multicasting allows a sending application to specify a multicast address and send one copy of the data to that address. This data is then distributed through the Multicast backbone to all the applications listening at that address. This reduces the network traffic and improves the performance.

Multicast is being used by FOS as a Release B service. Descriptions of this service are presented in section 4.2.4. This is presented for informational purposes. Some FOS testing with multicast service is expected to take place before Release B delivery.

Message Passing Service

The Message Passing Service allows for the exchange of information between applications running on different platforms. Clients send data to servers, which process the data and return the result back to the client. This interaction can be classified into three categories: synchronous, asynchronous, and deferred synchronous.

CSS will provide two implementations of Message Passing. The first model will provide for asynchronous and synchronous message passing - byte streams only - with store and forward, recovery and persistence. It will also include the concept of groups where a list of receivers belong to a group. A message sent to the group will be delivered to all the addresses registered in that local group. The second model will provide for asynchronous and deferred synchronous communication without recovery.

Both implementations are designed to take advantage of OODCE-provided DCE-Pthread class which is used to start and control the execution of a thread. The second mode requires more programmer involvement than the first model. Message Passing Service is generally intended to handle low volumes of data per message. Compare with k/ftp (below) for bulk data transfer.

Thread

A thread is a light weight process without the actual process overhead. Threads provide an efficient and portable way to provide asynchronous and concurrent processing, which is a requirement of network software. Threads can maintain thread specific data and can also share data with other threads in an application. This service provides functionality to create, maintain (scheduling, locking, etc.) threads. This service is described in more detail in section 4.2.5

Time

The Time Service keeps system (host) clocks in the ECS network approximately in sync by adjusting the time kept by the operating system at every host. This service changes the clock tick increments (rather than the actual clock) so that host clocks will be in sync with the some reference time provided by an external time provider. CSS will also provide a way to simulate time by applying a supplied delta time to the actual time. With in ECS, OSF DTS will be used to sync the system clocks. For more information please refer to section 4.2.6.

LifeCycle

Managing a system involves managing individual applications. An operator may want to start a new application, shutdown/suspend a running application due to anomalies. An application may not be active all the time to accept requests. In order to effectively use the CPU and memory it is desired to control the applications as well s some objects residing in the application by starting them on demand.

LifeCycle services can be broadly classified into two categories: Application and Object level. LifeCycle services for applications involve Startup, Shutdown, Suspend and Resume functionality on applications. This functionality lets the M&O manage server applications. MSS provides the application related LifeCycle functionality. CSS provides the internal APIs that are needed for the MSS to control the applications. LifeCycle services for objects provide the application programmer with the functionality to create and delete server objects residing in different address spaces.

Distributed Object Framework (DOF)

In an object oriented processing architecture, objects may be distributed in multiple address spaces, spanning heterogeneous platforms. The basic contract between an object and its users is the interface that the object provides and users can use. Objects can be spread across the network for reasons of efficiency, availability of data, etc. From the perspective of the requester of a service, invocation should be the same no matter where the object physically resides.

The distributed object framework will be implemented using OODCE. The set of core DCE services are naming, security, threads, time, rpc. In order to aid the application programmer, another layer of abstractions is provided with OODCE. Four generic classes: DCEObj, DCEInterface, DCEInterfaceMgr, and ESO will be available for application programmers to implement client-server applications.

Electronic Mail (E-Mail)

E-mail is a standard component of internet systems. It is useful for asynchronous, relatively slow notification of many different types. Also, E-mail is persistent, and will continue to try to deliver even if there are temporary network outages. The CsEmMailRelA class provides object-oriented application program interface (API) to create and send e-mail messages.

File Access-k/ftp

FTP is a internet standard application for file transfers. It allows a user to retrieve or send files from/to a remote server. The files transferred can be either ASCII or binary files. FTP also provides an insecure password protection scheme for authentication. KFTP builds on the standard FTP but adds a layer for strong Kerberos authentication. The CsFtFTPReIA object provides an object for managing FTP sessions between clients and servers to allow programmers to transfer files between machines.

Bulletin Board

Bulletin Board is another internet standard application, however unlike e-mail, Bulletin Board messages are directed to all readers of a named group. It uses the Network News Transfer protocol (NNTP) for sending and receiving messages. The CsBBMailReIA object provides object-oriented application program interface to send e-mail.

Virtual Terminal

Virtual Terminal access refers to remote log on to a machine. This software is provided on all platforms. The server telnetd will listen to incoming telnet clients and will allow remote logons. There is also a secure version of telnet and telnetd using Kerberos authentication which CSS will provide where available. This service is allowed only within ECS due to security considerations.

X is a Graphic User Interface conforming to the X/Open standard. While X is not a specific CSS Release A service this description is listed here for informational purposes. It consists of a client and a server where the client displays the actual interface. Developing applications in X is cumbersome and complex. OSF Motif is another standard, layered on top of X which provides a high level application programming interface to make the application development easier. Applications developed with Motif will work with an X server. The X client/server connection presents some significant security risks; therefore ECS will not support applications where the X client and the X server reside on different platforms. Users can down load data from ECS and can use the X application to view the data on their local machines. Alternatively, the program should consider providing dedicated circuit access from a user client to connect to an ECS X application.

Event Log

Event log provides the programmers the capability to record events in to files. Events are broadly classified into two categories: management events and application events. Each event is recorded with all the relevant information for identifying and for later processing. Management events needs to be recorded in a history file and on some occasions reported to the Network Node Manager. Application events are only recorded into a programmer specified file. Event log provides a uniform way for the application programmers to generate and report (record) events. For further information please refer to the Event Log section.

This page intentionally left blank.

4. Release A Design

4.1 Introduction

This section gives an introduction of all the CSS services whose directives and guidelines/samples are provided in section 4.2. Each of the individual CSS services such as Directory Naming, Security, Message Passing, Thread, Time, etc. are described with an Overview, a Context, Directives and Guidelines, Sample APIs and an Object Model description of the classes for the service. Table 4.1-1 provides a road map for the CSS Developer's Guide section.

Table 4.1-1. CSS Design Section

Services	Section
Object Services	4.2
Directory Naming	4.2.1
Security	4.2.2
Message Passing	4.2.3
Multicast	4.2.4
Thread	4.2.5
Time	4.2.6
LifeCycle (Initialization / Activation / Deactivation)	4.2.7
Distributed Object Framework	4.3
Common Facility Services	4.4
Electronic Mail	4.4.1
File Access	4.4.2
Bulletin Board	4.4.3
Virtual Terminal	4.4.4
Event Logger	4.4.5

Distributed Computing Environment (DCE) from the Open Software Foundation (OSF) is a CSS baseline COTS product for release A. DCE supports for client/server applications development in a heterogeneous platform environment (including SUN, IBM, HP, DEC, Silicon Graphics (SGI), Cray), and is available from multiple vendors (including HP, IBM, DEC, Transarc, SUN, SGI). DCE provides the underlying COTS for many of the services described in this CSS section (including directory, time, threads, security {Kerberos implementation}, and RPCs in a unified environment. Release A's communications subsystem (CSS) encapsulates DCE so as to simplify the change from DCE to CORBA in the future.

A COTS product, OODCE from HP was selected as the encapsulation method. OODCE provides an object-oriented layer on top of DCE by providing a set of class libraries and an Interface Definition Language Compiler called IDL++ to generate stub code in C++ language. With OODCE comes the ability for the applications developer to use object orientation in their client/server development, and use C++ class libraries. The encapsulation method also entails CSS development of custom APIs as the interface for application developers to access OODCE and DCE functionality. Note that each CSS service (which application developers need to interface with) provides service specific DCE-encapsulating APIs. These custom APIs are intended to reduce breakage when ECS inserts CORBA 2.0 or other future technology into a later release. This custom API/OODCE/DCE combination thus forms the foundation of the CSS migration strategy to CORBA.

It should be understood that a number of the mandatory CSS services are normally used only when a client/server session is being established (find and bind from the directory service, authentication and authorization from the security service). Thereafter client and server can agree to other interfaces (e.g., DBMS client-to-DBMS server native protocol).

Like other subsystems, CSS depends on the MSS agent that runs in all ECS provided workstations and servers to provide management status and event notification of the CSS servers and daemons to MSS. In turn CSS provides MSS the CSS e-mail, ftp, event logging, and message passing services for its processing requirements.

4.2 Services Description

4.2.1 Directory Naming Service

4.2.1.1 Overview

The Naming Service is one of the fundamental facilities needed in distributed environments to uniquely associate a name with resources/principals along with sufficient information so they can be identified and located by the name even if the named resource changes its physical address over time. Naming is used primarily by service providers to register information about a service and by clients to locate the services.

Naming may be used more generally to store and retrieve any general information that is required to be made available about an object across a network. This information could include a server's binding information (e.g., an ECS search program that is going to search the databases for a specified criterion), file set locations (a file containing the forest vegetation for a specific time), an science product type (e.g., MODIS 2B), a network resource (e.g., a printer), information about principals (the security namespace containing user passwords, telephone numbers). The Naming service organizes this information in namespaces.

There are two widely known Name service specifications: ISO X.500 and the Internet's Directory Name Service (DNS). DCE Global Directory Service (GDS) is an implementation of the X.500 specification, and BIND is an implementation of the DNS. Of the two, BIND is more widely used. While these are standard enterprise namespaces, the interface provided for these namespaces is different and is at a very low level for the application programmer to use. X/Open

had addressed the need for communication across namespaces with an interface called X/Open Federated Naming (XFN) that specifies a common interface a namespace has to support. The intent of this specification is to provide a common abstract interface that can be implemented on top of both the DNS and X.500 as well as to provide a way for the enterprise namespaces to communicate with each other.

CSS will provide an implementation of both the DNS and the X.500 namespaces. These namespaces are used to connect the local namespaces with other namespaces. CSS will provide OSF Cell Directory Service (CDS) and OSF GDS as the local namespaces. These local namespaces are used to store server binding information. Both CDS and GDS provide a standard (X/Open) application program interface called the XDS/XOM interface for the application programmers to interact with them. While namespaces are primarily used to save server information by application frameworks (like the Distributed Object Framework), they do not normally use this interface to communicate with the namespace. A specialized, more efficient internal interface is provided for these application frameworks by the local namespace (like the NS interface for CDS) to store and retrieve server binding information.

Application programmers need to use the XDS/XOM interface in order to store and retrieve application specific information into the namespaces. XDS/XOM interface is too tedious and complex to use. CSS will provide an XFN like interface to store and retrieve information in the CDS and X.500 conformant namespaces (OSF GDS is an X.500 conformant namespace). The XFN functionality will be implemented on top of the XDS/XOM interfaces. As such, the functionality provided here should work on other X.500 namespaces.

There are standard ways to startup and shutdown the DNS and the X.500 namespaces which will be provided to the M&O staff by the MSS Management applications. Starting and shutting down CDS is part of the Distributed Object Framework and is provided to the M&O staff through the MSS Management applications.

The CDS and GDS consists of entries (name and attribute value pairs). These entries may be protected through Access Control Lists (authorization). While CDS provides complete Access Control of these entries, GDS provides only a limited Access Control (authorization) which is better than UNIX OS Access Control Lists.

A name consists of a sequence of one or more contexts composed according to the naming convention, and the entry name. Each entry name is associated with a set of zero or more attributes. Each attribute in the set has a unique attribute identifier, an attribute syntax, and a set of zero or more distinct attribute values.

4.2.1.2 Context

Directory Naming is an infrastructure key mechanism and is used by ECS subsystems who need to use a Cell Directory namespace as a database to enter or retrieve information stored in the form of attribute-value pairs.

M&O will use an MSS application to store user profiles containing such information as a user's telephone number and office location and retrieves it. FOS Planning and Scheduling will use the namespace to save process related information such as location and messaging interest. Other

applications may use the namespace to store and retrieve any data that should be location independent and be visible to several applications such as a common message queue to send messages asynchronously, a Universal Resource Locator (URL) of an object that an SDPS application uses. All of this interaction with the namespace is done via the interface provided by the CSS Naming service. Other interaction with the namespace to save and retrieve server binding information is done via the local specialized interface.

4.2.1.3 Directives and Guidelines

Directives:

The programmer must do the following in order to store/retrieve information from the DCE Namespace:

- Log in to the Cell Directory Namespace.
- Have read/write access to the directory path where entries will be added, modified, or deleted via the API. There is a command, the rgy_edit command, that the cell administrator must run, in order to assign rights to a particular object in the CDS.
- Create a context root name. Instantiate an EcDnContext object. The context class is used to define the type of name to be used in the DCE environment, it takes either a global root name(/...) or a cell root name (/.:) A context object is passed to the Composite Name class as an ordered sequence of components.
- Create subordinate contexts (EcDnContext type of objects).
- Instantiate an EcDnCompositeName object and add the contexts to it. The composite class allows the user to form a composite name (by means of the AddContext operation). A composite name will represent a full path name (equivalent to a directory) or an entry in a database (equivalent to a file). The user can perform various operations on a composite name when the composite name is a directory such as list the contents, read entries, add and delete elements (an attribute/value list pair) element into this directory, and get the element types that had not been previously used.
- In order to add or delete an element to the composite name, must construct an element object using the element class. The element class provides methods to add values, get value list, delete values, modify values, and to get the element name.
- The element class makes use of the value and attribute object classes. The attribute class contains the attribute name and type, and provides methods to obtain the attribute name and type. The value class contains a value.

Guidelines:

The Directory Naming Service is used to construct large, enterprise-wide naming graphs. A name-to-object association is called a 'name binding'. It is defined relative to a 'Naming Context'. A 'Naming Context' is an object that contains a set of name bindings in which each name is unique. 'Naming Contexts' represent 'directories' or 'folders' and other names identify 'document' or 'file' kind of objects.

The Context here can be thought of as a database key and the attributes are the actual information associated with the key. Each context forms a hierarchy where each context has a parent. Each namespace has a parent that may be another namespace maintained elsewhere. There will be a root context, with no parent, through which all the contexts can be accessed.

Leaf contexts form the key to an entry in the namespace, to which information is attached. This information contains a list of attributes, each of which contain an identifier, a syntax representing the type of the attribute, and a list of values for that attribute. Each attribute type is identified by a unique number in the namespace. These unique numbers will be obtained from standard bodies so that there won't be any conflict in the unique ids used across different namespaces.

CSS Name service provides wrapper functions to map some of the XFN calls to the underlying namespaces that are supported: CDS/GDS. These wrappers are written on top of XDS/XOM interfaces. Since XDS/XOM are X/Open's standard interfaces to X.500 namespaces, the functionality provided by CSS will work on other X.500 namespaces. CSS will only support features that are supported in the underlying namespace. For example, not all namespaces support searching in the namespace, as such, a search wrapper function will not do anything when acting on such a namespace. Both the GDS and DNS are replicated and distributed namespaces.

CSS provides five custom classes that can be used by applications to utilize the Cell Directory Name Space as a database. CSS divided the naming structure into two parts: a context and a list of elements. Each element represents an attribute-value list pair. Figure 4.2.1.3-2 pictorially describes the CDS entry structure.

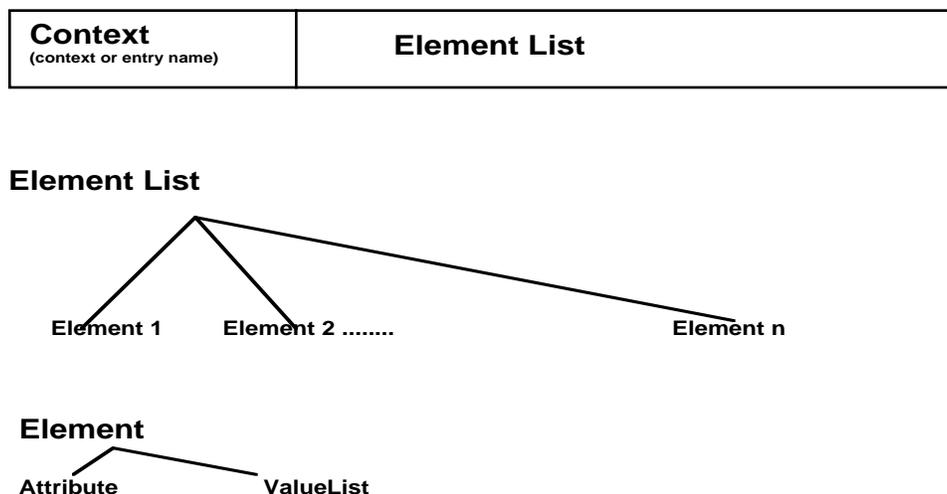


Figure 4.2.1.4-2. Naming Service - CDS Entry Structure

If the user wants to see graphically the contents of the Directory Naming Service, the CDS Browser is available. The CDS Browser (a GUI interface) allows a user to view the contents and structure of the cell namespace. The Browser can both display an overall directory structure, and

show the contents of directories. It can also be customized to display only a specific class of object names.

Since the CDS Browser is not available on all the platforms, the 'cdscp' command can be used to view the contents of the cell namespace. 'cdscp' allows one to view selected contents of one directory at a time. It is an interactive command, not a GUI interface. For example, to display all objects in the directory '././common/dev/store', the user can enter the following:

```
cdscp=>l obj ././common/dev/store/*
```

For more information regarding CDS Browser or the 'cdscp' command, please refer to the DCE administration guide.

Regarding the practical limit on the size of a DCE cell, it is still a bit early in the product life to have substantial experience with large-scale DCE installations. But there are some large cells in operation. Certainly it is reasonable to plan on cells with at least thousands of nodes and perhaps tens of thousands of users.

The University of Michigan Center for Information Technology Integration has done a study in which they added 50,000 entries to the Cell Directory and to the security registry. Their results are reported in technical reports 93-12 and 94-1.

Regarding how much memory and disk space is required for DCE services, this depends on the size of the cell, the number of users, number of services, etc. According to a paper present by Dan Hamel of Transarc, at the Decorum conference in February 1994, the following can be used as rough guidelines:

- Security server: 2k per principal/account; same at replicated sites
- Directory server: 10k per directory, 1k per object; same at replicated sites
- End-user machines: Each dce_login creates new credential files, which can build up. Space usage can range from less than 1k to over 100k.

Regarding whether a machine can be a member of more than one DCE cell, this is not possible at present. A machine can only be in a single cell under DCE 1.0.3. However, it will be possible for cells to cooperate when DCE 1.1 is deployed. See the next question.

Present Available

- Implementations of both DNS and X.500 are available. DNS/BIND is available in public domain from the Internet at no cost. CSS will provide this public domain BIND namespace. Implementations of X.500 are available as COTS products. GDS is one such implementation that comes as bundled software with DCE. Since DCE (OODCE) is the chosen infrastructure, CSS will provide GDS as the X.500 conformant namespace. Both BIND and GDS are replicated and distributed namespaces that support local client caching. They both provide some degree of security.

Future Available

- X/Open is currently developing code to support the XFN interface. This standard will be implemented in Release B, provided that it is ready in that time frame. In Release A, CSS will develop the required XFN functionality (with the exception of search) on top of the X/Open XDS/XOM interface for the X.500 namespaces.

4.2.1.4 Sample Application Programmer Interface

Following are the methods the application's developer uses to store/retrieve/list elements in a namespace.

Sample #1

List the contents of a Directory Service

Description

In order to create a composite name the user enters a series of atomic names that are in accordance with the CDS naming conventions. First create a context root name. The developer must define the type of name to be used in the environment, either a global root name or cell root name. Second, create the subordinate contexts, instantiate an object of `EcDnComposite` type and add the contexts to the composite name.

Then, proceed to list the entries/directories in that composite name; using the composite name object call the method 'listCtx'.

```
// Create a context root name

EcDnContext *ctxP = new EcDnContext(1);           // where 1 is for ./:

// Create subordinate contexts

EcDnContext *ctx_1P = new EcDnContext("hp", 0); // 0 indicates "hp" is a directory
                                                    // not a file

EcDnContext *ctx_2P = new EcDnContext("examples", 0); // 0 indicates
                                                    // "examples" is a
                                                    // directory not a file

// Instantiate an object of type EcDnCompositeName and add the contexts to it

EcDnCompositeName *comp_nameP = new EcDnCompositeName(ctxP);

status = comp_nameP->add_ctx(ctx_1P); // we now have ./:hp

status = comp_nameP->add_ctx(ctx_2P); // and now we have ./:hp/examples
```

```

// List the Directory Service

status = comp_nameP->list_ctx(&lstP);    // Obtain contents of ./hp/examples

                                         // output from list_ctx is lstP which
                                         // is a pointer to a linked list of
                                         // containing the immediate
                                         // subordinates of ./hp/examples

```

Sample #2

Add an element into the Directory Service. The user defines the entry name, type and its respective attribute name, type and value. Makes a request to add an element.

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Proceed to add the element. Instantiate an attribute, instantiate an element, create and instantiate values, add values to the element, and add element into the Directory Service. Using the composite name object call the method 'addElement'.

```

// Create context root name

EcDnContext *ctx_AP = new EcDnContext(1);    // where 1 is for ./

// Create subordinate contexts

EcDnContext *ctx_1AP = new EcDnContext("hp", 0);    // 0 indicates "hp" is a
                                                    // directory not a file

EcDnContext *ctx_2AP = new EcDnContext("examples", 0); // 0 indicates
                                                    // "examples" is a
                                                    // directory not a file

EcDnContext *ctx_3AP = new EcDnContext("sleeper", 1); // 1 indicates "sleeper"
                                                    // is a file not a
                                                    // directory

```

```

// Instantiate an EcDnComposite name and add the contexts to it
EcDnCompositeName *comp_nameP = new EcDnCompositeName(ctx_AP);
status = comp_nameP->add_ctx(ctx_1AP);           // we now have ./:hp
status = comp_nameP->add_ctx(ctx_2AP);           // add on and get
                                                    // ./:hp/examples
status = comp_nameP->add_ctx(ctx_3AP);           // final name is
                                                    // ./:hp/examples/sleeper

// Instantiate an attribute
EcDnAttribute *attr1 = new EcDnAttribute("CSSAttr"); // new attribute

// Instantiate an element
EcDnElement *elt1 = new EcDnElement(attr1);       // This element is for the
                                                    // new attribute "CSSAttr"

// Create/instantiate values
EcDnValue *val1 = new EcDnValue("CSSValue1");    // 1st value for "CSSAttr"
EcDnValue *val2 = new EcDnValue("CSSValue2");    // 2nd value for "CSSAttr"

// Add values to the element
status = elt1->add_value(val1);                   // Add the values into the element
status = elt1->add_value(val2);                   // for the new attribute "CSSAttr"

// Add element in the Directory Service
status = comp_nameP->add_element(elt1);           // now add the element to
                                                    // ./:hp/examples/sleeper

```

Sample #3

Read elements from the Directory Service.

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Proceed to read an element. Using the composite name object call the method 'getValueList'. This operation will return the element information (the list of attributes and its values).

Delete the value object from the element and the element from the composite name. Deallocate any memory used.

```
status = comp_name->read_element(&elt_listP);    // Composite name contains
                                                // /./hp/examples/sleeper
                                                // Output from read_element is a
                                                // pointer to a linked list of
                                                // element objects
                                                // (attributes and their values)
```

Sample #4

This scenario describes how to obtain a list of values pertaining to an element (attribute).

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Given an element, specify the particular attribute to be read. Using the element name object call the method 'getValueList'. This operation will return the a list of the attribute values.

Delete the value object from the element and the element from the composite name. Deallocate any memory used.

```
status = elt1->get_value_list(&val_listP);      // elt1 points to a specific
                                                //element for which we want
                                                // to obtain all the values
                                                // Output will be val_listP
                                                // which is a pointer to a
                                                // linked list of value objects
```

Sample #5

Delete a value from the element

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Given an element delete its value.

```
status = elt1->delete_value(val_obj);           // Delete the value object
                                               // from the element
```

Sample #6

Delete element from the Directory Service

Description

Create a context root name. Define the type of name to be used in the DCE environment, either a global root name or cell root name. Then, create the subordinate contexts, instantiate an object of EcDnComposite type and add the contexts to the composite name, including the entry name. Delete the given element.

```
status = comp_name->delete_element(elt1);      // Delete element elt1 from
                                               // the composite name -
                                               // ./hp/examples/sleeper
```

4.2.1.5 Object Model

Table 4.2.1.5-1 summarizes the Directory Naming Service classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.1.

Table 4.2.1.5-1. Naming Service Object Responsibility Matrix

Class name	Description
EcDnContext	The EcDnContext class defines the path/set of bindings with distinct atomic names. Every context has an associated naming convention. A context object is passed to the EcDnCompositeName object in a structural form as an ordered sequence of components. It is the key part of the entry in a namespace that is used to uniquely identify it.
EcDnCompositeName	The EcDnCompositeName class defines a composite name, which is a nested set of contexts in a given hierarchy concatenated together to establish a Directory Service path name. This class provides functionality to create/modify/maintain context part of an entry in the namespace. It provides methods to concatenate contexts, list the contents of the composite name in the Directory Service (soft links/object entries), read entry names, add elements (attribute/value list pair), read element information, and delete elements.
EcDnElement	The EcDnElement class contains an element, which is an attribute-value list pair. It is referenced by the EcDnCompositeName class. This class provides methods to add value(s), get value list, delete value(s), modify value(s), and get the element name.
EcDnAttribute	The EcDnAttribute class contains an attribute name, and type. It is also referenced by the EcDnElement class. This class provides methods to get the attribute name and type.
EcDnValue	The EcDnValue class contains a value. It is also referenced by the EcDnElement and EcDnAttribute class.

4.2.2 Security Service

4.2.2.1 Overview

In distributed systems applications rely on services provided by servers running in different address spaces on heterogeneous platforms. Servers are independent and their main functionality is to listen for client requests, process the request and send the results back to the clients. This division of processing can be done for any number of reasons such as efficiency, data availability, etc. In addition to a client invoking a request, and the server processing that request, both the client and the server may need to use mechanisms to protect resources as well as the integrity of the data exchanged. These mechanisms are authentication, authorization, tamper-proofing (for data integrity) and encryption (for data privacy). While authentication should always be used in every conversation between a client and a server, the mechanisms for authorization, data integrity and privacy are based on security policies of the system(s) and the application-specific need for those mechanisms. These concepts are explained in detail in the CSMS requirements document (DID 304). This document explains how these mechanisms are achieved in the current design.

Authentication is the process of verifying the validity of a principal. Authentication is usually done at two points. Initially when users login to the ECS domain, authentication is done by a "trusted third party" who supplies server's credentials for principals to use with application servers .

Authorization is the processing of deciding what sort of users/groups should be allowed to access what services/resources and then allow/deny the service. In authorization, each resource is associated with a list of permissions that should be granted to different kinds of user and different kinds of access operations. This is used to selectively to grant certain principals access to some resources. Authorization is performed by Access Control List (ACL) mechanism. An ACL is an entry with information such as the name of the user/group and the permissions list associated with them. which indicates the kind of permission/s given for the user/group. ACLs should be created and maintained for all the application specific objects.

When data is transmitted over the network from one application to another, the integrity of the data should be preserved. This is to make sure that the copy of the data the receiver gets is exactly same as the data that the sender sends. This tamper-proofing of the data may or may not be used with methods for protecting the privacy of the data. Checksums or secure hashes are often used to guarantee data integrity.

Encryption is the process of encoding a message into cipher text using a key. The process of decoding the cipher text to its original form using a key is called decryption. Encryption is used to maintain the privacy of data transmitting over the network.

While authentication is necessary, other features; authorization, checksums and data privacy are optional. ECS application can selectively choose to use these security features depending on their specific needs.

4.2.2.2 Context

It is assumed that all ECS services (but not necessarily all clients) run in the DCE/OODCE environment. M&O needs to be able to create and maintain accounts for principals to login to ECS hosts. The authentication service is used by the ECS Login service and may be used by both the SDPS/FOS clients and servers to authenticate each other. The authorization service may be used by any server to protect access to a resource, such as allowing only the authorized Principal Investigator to browse through some instrument data. Data integrity may be used by in IST sessions to make sure that the command requests received from Instrument Investigators are not modified in transit. Encryption may not be needed by most ECS applications directly, but is used internally by the other three security services.

4.2.2.3 Directives and Guidelines

The OODCE COTs product will be used for the security functionalities along with the following CSS provided custom software:

- EcsSecurity
 - Provides a layer encapsulating the underlying security classes such as EcsAcl, EcsAclDb, EcsModifyableAcl, EcsAclStorageManager, DCEAclMgr, DCEAclSchema and DCESecId.
 - Application developers should be able to create ACLs for application specific objects by using this class. When acls are updated and the application server goes down for some reason, when it is brought up again, the acls available are the initial acls and not

the updated acls. In order to have the latest updated acls available for use by the application server every time it is brought up, this persistent storage is provided. Everytime acl is updated it is written into persistent storage file and whenever application server is brought up, the updated acls are written from the persistent storage file back into the memory for use by application server. For this functionality it is required to implement the methods of the following classes: EcsAcl, EcsModifyableAcl, EcsAclStorageManager and EcsAclDb.

- EcsFilePassword
 - This inherits from the COTs provided class DCEPassword. During login context, in case of interactive principals, password is supplied on command line. The DCEPassword class takes a keyfile name as one of the arguments and provides non-interactive principals with their passwords to be stored in the keyfile, so that non-interactive principals (servers) can establish their identity without doing a login.

The server keytab file and the persistent aclfile should be placed in the same directory where the database server is going to be executed. It is required to give read permission to the server principal for the serverKeyTabFile, read and write permission for the persistentAclFile. Both these files should be placed in the same directory where the database server will be executed. The persistentAclFile will always have the latest Acls of objects.

Besides, the following COTS provided operator interfaces can be used for creating and maintaining user accounts and Acls, create server keytab file, etc. :

- Operator will use acl_edit to modify permissions to principals to access different services.
- Through rgy_edit operator will create the server keyTabFile to give an application server principal its own identity during its login context.
- Through rgy_edit, operator creates user accounts.

Section 4.2.2.4 provides examples to demonstrate why and how each of the classes and the methods should be used by an application developer. Section 4.2.2.5 gives a brief description of all the COTS and CSS provided security classes. For more information regarding the COTS utilities, refer to the DCE administration guide.

4.2.2.4 Sample Application Program Interface

Sample #1

Setting Client/Server Authentication Preferences

Description

For a secure communication, the first step an application developer has to take, is to set the following authentication preferences for both the client and the server objects:

Authentication Protocol - Authentication protocol can be DCE shared-secret key authentication, where the server gets its password from a keytab file for establishing its login

context, or no authentication, where no tickets are exchanged, or DCE default authentication service (The current default authentication service is DCE shared-secret key.), or the DCE public-key authentication (which will be supported by DCE 1.2). This is specified by the server (per process) to indicate the type of authentication it is OFFERING, and by the client (on a per object basis) to indicate the type of authentication it DESIRES to have.

Authorization Protocol - Authorization protocol can be either No authorization, where the server performs no authorization, or Name-based, where the server performs authorization based on the client's principal name, or PAC/DCE based, where the server performs authorization using the client's DCE Privilege Attribute Certificate (PAC) sent to the server with each RPC made with *binding*. The type of authorization protocol is specified by the client to indicate the authorization type DESIRED by the client.

Protection Level - Protection level specifies the protection level for RPCs made using *binding*. It determines the degree to which authenticated communications between the client and server are protected by the authentication service specified by authentication protocol. The protection level when selected to be "packet_integrity" will ensure data integrity (i.e. ensures that data is not modified during transit) by adding encrypted checksums to the data. Also specifying the protection level as "packet_privacy" will ensure the privacy of data through the use of secret-key encryption. However in trying to achieve data integrity/privacy there is a tradeoff. i.e. more restrictive the protection level, the greater the negative impact on performance.

i. The appClientObj (**per object basis**) must invoke the *SetAuthInfo()* API as follows:

a. Include files

```
#include "appC.H" // Generated by application IDL.
```

b. Construct an instance of the Client Class

```
appClientObj appCObj;
```

c. Invoke the SetAuthInfo API

```
appCObj.SetAuthInfo(  
  
(unsigned_char_t*) princName, // Name to identify server principal.  
  
rpc_c_protect_level_pkt_integ, // Specifies packet level protection which is the  
// highest level guaranteed to be present in the RPC  
// runtime.  
  
rpc_c_authn_dce_secret, // Specifies DCE secret-key authentication protocol.  
  
(rpc_auth_identity_handle_t) NULL, // NULL specified to use the default security login  
// context for the current address space.  
  
rpc_c_authz_dce); // PAC/DCE based authorization protocol specified.
```

ii. The ESO (DCEServer object 'theServer', already defined in the DCEServer.C file) should invoke the SetAuthInfo() (**per process**) as follows:

a. Include files

```
#include <oodce/Server.H>

#include "appS.H" // Generated by application IDL.
```

b. Invoke SetAuthInfo

```
theServer->SetAuthInfo(

(unsigned_char_t*) princName, // Specifies the Principal Name to use for the Server
                             // when authenticating RPCs.

rpc_c_authn_dce_secret, // DCE secret-key authentication protocol specified.

(void*) keyFile); // Specifies the 'KeyFile' where server gets
                  // password.
```

Sample #2

Key Management and Login Context

Description

Next is the key management and login context part. The authentication mechanism is based on two fundamental constructs: *principal identities* and *secret keys*. The basic authentication policy issues therefore have to do with how applications manipulate these data: how they acquire their principal identities and how they maintain the security of the secret keys (i.e. In a network environment, when principals want to access the resources over the network, how will they provide the encrypted password, etc.).

When first invoked, a server process uses the login context of the user who invoked it as its principal identity. This may be sufficient for the application's purposes; however it may need to assume its own identity.

The server assumes its own identity by retrieving its secret key, which is analogous to a user's password, and using the key to establish its own login context. The server's key is stored in two places: by the server in a local key data file; and by the Security Service in its Registry Service database. Keys for servers that require root access to file system data or for servers that need to run as root are stored in the system-wide key file which is owned by root. Servers that do not need to run as root should store their keys in a private file, which the server has access to but nobody else does. The server's local copy is used by the server runtime to decrypt incoming client tickets, and is also by server to acquire its own login context.

The server will establish its password and login context as follows:

a. Include files

```
#include <oodce/Password.H>

#include <oodce/Login.H>

#include "EcsFilePassword.H"
```

b. Establish Server Password

(The EcsFilePassword class inherits from the base class DCEPassword provided by OODCE)

```
EcsFilePassword passwr(serverPrincName, serverKeytabFilename);
```

c. Establish Server Login Context

(DCEStdLoginContext is the default implementation of the DCELoginContext class defined by OODCE)

```
DCEStdLoginContext loginCntxt(&passwr);
```

Sample #3

Reference Monitor

Description

The ServerProg (server main program) will create a DCERefMon object. The reference monitor is used to provide mutual authentication (that is, allow the server to check that the client is as claimed) and to ensure that the server is willing to meet the client preferences for protection and authorization.

It can perform basic checks before any application code is entered. In general, these checks are as follows:

- Is the client program authenticated (i.e., has the client principal established a login context?)
- Does the protection level requested by the client meet the requirements of the server program?
- Does the authorization model requested by the client meet the requirements of the server program?

Instantiation of DCERefMon object is as below:

a. Include files

```
#include <oodce/RefMon.H>
```

b. Instantiation of the RefMon Object :

```
DCEStdRefMon *thisRefMon = new DCEStdRefMon(protectLevel, authnSvc,  
                                             authzSvc);
```

Sample #4

Generating Object UUID

Description

The ServerProg creates a DCEUuid object and initializes it with an object uuid created using uuidgen command (Refer to DCE manual for details on uuidgen) as follows:

a. Include files

```
#include <oodce/Uuid.H>
```

b. Instantiate DCEUuid object

```
DCEUuid      objectUuid("34c53cfa-9b3d-11cc-adaf-080009627155");
```

Sample #5

Initialization of AclManagement

Description

The ServerProg creates DCEAclMgr object through DefineAclMgr macro. The DCEAclMgr class allows server developers to register a rdacl interface manager object with the global DCEServer object. There is one instance of DCEAclMgr per application server, which is referred by a global reference called acl_manager. Creation of the DCEAclMgr global object is as below:

a. Include files

```
#include <oodce/AclStorageManager.H>
```

```
#include <oodce/AclMgr.H>
```

```
#include "appS.H" // Generated by application IDL.
```

b. Instantiate DCEAclMgr object

(Note: The DCERefMon object must be instantiated before calling the DefineAclMgr macro.)

```
DefineAclMgr(*thisRefMon, objectUuid, "database server");
```

Here the first argument DCERefMon object will enforce the security policy for requests coming through the rdacl interface. The second parameter, an object uuid will be registered with CDS and with the endpoint mapper to enable acl_edit tool to contact the right server. The third parameter is the server name used to identify this instance of a DCEAclMgr.

Sample #6

Acl Management

Description

The concept of Access Control Lists (ACLs) is used to perform authorization. DCE/ OODCE provides a set of mechanisms for access controls, which include;

- The authenticated identity and privilege attributes (in the form of credentials) of service requesters, provided by the RPC runtime to servers.
- ACLs which servers may associate with objects they control.
- A default mechanism for determining a service requester's privileges from an ACL and the requester's credentials.
- Tools for administering ACLs.

Create an instance of EcsSecurity object. Through this object invoke functions for performing CreateAclSchema(..), GetAclSchema(..), CreateAcl(..), CreateAclDatabase(..), etc. as follows:

a. Include files

```
#include <oodce/AclSchema.H>
```

```
#include <oodce/AclDb.H>
```

```
#include "EcsSecurity.H"
```

b. Instantiate EcsSecurity object

```
EcsSecurity *EcsSecurityObj = new EcsSecurity(char *databaseName);
```

c. Create acls, acl schema and acl database

Through this object can invoke the following functions:

```
EcsSecurityObj->CreateAclSchema();
```

```
ECSAclDb *EcsSecurityObj->CreateAclDatabase(databaseName, *_Schema, char  
*persistentDbName);
```

```
EcsSecurityObj->CreateAcls();
```

Sample #7

Check on Client's Authorization Privileges

Description

The application server is brought up and it should be running. The application client when run, will be requesting some permission to a resource (as indicated earlier a resource can also be an idl implemented method). The application server will check the ACL associated with the object/resource and compares it with the client's PAC and makes a decision about the client's

requested access to the resource. If authorized the client will perform the operation else an exception is thrown by the server on to the client's side.

Invoking the `IsAuth` function within the `appServerObj` to check client's authorization privileges:

```
class EcsAcldb;
EcsAcldb *_database;
if(_database->IsAuth("EcsAppObj")
    appMethodX(...);
else
    traceobj << "Not Authorized to Perform appMethodX\n";
```

If the requesting client has permission for *appMethodX*, `IsAuth` returns `TRUE`, otherwise it returns `FALSE`.

Application developer creates `serverkeytab` file using `rgy_edit` utility (explained in one of the scenarios below). Also when the server is still up and running, the "`acl_edit`" client can be run in order to manipulate ACLs (described in one of the below scenarios).

Sample #8

Using the COTS provided operator interface utility "`rgy_edit`", to create server keyfile.

Description

The application developer invoke the `rgy_edit` program and run the `ktadd` command providing the server principal name and the server keytab file name as "`ktadd -p SrvPrincName -f SrvKeyFileName`". The utility prompts for a password "Enter password:" twice. Supply the password both times. Exit `rgy_edit` utility. Give the server principal at least `READ` permission to this file "`SrvKeyFileName`" (This is important). Now the server principal is able to establish its identity during login by getting its password from this `keyTabFile`. For more information, refer to `DCE Administration Guide`.

Following is an example as to how you can run **`rgy_edit`** utility for creating a server keytab file:

- a. Login to dce as an authenticated user.
- b. Running `rgy_edit` Utility
 - i. Enter `rgy_edit` by typing **`rgy_edit`**
 - ii. You get **`rgy_edit=>`** prompt.
 - iii. At the above prompt type the following command to create the **`srvKeyFile`** :
`ktadd -p srvPrincName -f srvKeyFile`
 - iv. You get **`Enter Password:`** prompt to enter password

- v. At the above prompt supply the **srvPrincPassword**.
- vi. You get **Re-enter password to verify:** prompt for password verification
- v. Supply **srvPrincPassword** again.
- vi. At this stage the **srvKeyFile** is created. Verify if the **srvKeyFile** is created (OPTIONAL) with the following command:

ktlist -f srvKeyFile

- vii. If the **srvKeyFile** is created, you get the following message:

/.../cellName/srvPrincName

(indicating the owner of the **srvKeyFile** created to be **srvPrincName**.)

else you get the following message:

Unable to retrieve(s) - Specified key table not found

- viii. Exit out of **rgy_edit** utility, by typing **q** OR **e**.
- ix. Provide the **srvPrincName** with at least READ permission to the **srvKeyFile**. (IMPORTANT)

NOTE: For more help refer to DCE Administration Guide and also when you are in **rgy_edit** type **h** for help.

Sample #9

Using COTS provided "acl_edit" utility, to manipulate Acls.

Description

M&O staff invoke the **acl_edit** program and then identify a resource (the access method associated with a database server application) to view the ACL associated with the resource. The **acl_edit** program calls the database server program to get a printable representation of the ACL associated with the access operation. The database servers view operation checks if the client can perform the view operation and returns the ACL associated with the access operation in a printable format. **acl_edit** then displays this on the screen. The M&O staff then invoke the edit operation to edit the selected ACL. The **acl_edit** invokes the read **acl** operation to get a copy of the **acl** associated with the access operation. Upon checking the permissions associated with the read operation, a copy of the ACL is sent back to the **acl_edit**. The M&O staff change this ACL and invoke the save operation. The **acl_edit** then invokes the replace **acl** operation of the database server passing this modified ACL. The replace operation after checking the authorization privileges, replaces the supplied ACL in memory and may update the contents onto the disk.

Following is an example as to how you can run **acl_edit** tool to manipulate ACLs:

- a. Login to **dce** as an authenticated user with proper privileges (ex: privilege to modify an object's ACL)

b. Run the application (which manages the object/s for which you want to run `acl_edit` for) server executable.

c. Run `acl_edit` for object (say) **obj1**

i. Type the following command:

```
acl_edit ./:/cdsDirName/appSrvCdsName/obj1
```

ii. You get `sec_acl_edit>` prompt

iii. View the object's ACL list by typing **l**. You see the ACL similar to the following example:

```
# SEC_ACL for ./:/cdsDirName/appSrvCdsName/obj1  
# Default cell = /.../edfcell.hitc.com  
unauthenticated:r  
group:CSS:rtx  
user:manand:rwcd  
any_other:rt
```

iv. Modify ACL. Following are some examples.

```
Example 1 : m user:joe:rwta
```

(adding a new entry for user joe in the above shown example)

```
Example 2: m group:CSS:rtxa
```

(modifying the ACL entry for group CSS users)

v. Commit the above modifications by typing **co**

vi. View the modified ACL list. It will now look as follows:

```
# SEC_ACL for ./:/cdsDirName/appSrvCdsName/obj1  
# Default cell = /.../edfcell.hitc.com  
unauthenticated:r  
group:CSS:rtxa  
user:manand:rwcd  
user:joe:rwta  
any_other:rt
```

vii. Exit out of `acl_edit` by typing **e**.

NOTE: For more information on `acl_edit` utility, refer to DCE reference manuals and type `h` (for help) when you are in `acl_edit`.

4.2.2.5 Object Model

A brief description of all the CSS customized security objects as well as the COTS provided classes used by the CSS security is provided in the table 4.2.2.5-1 below. For more information on any of the COTS provided objects, please refer to the OODCE reference manuals. CSS customized security classes are discussed in detail in the Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.2.

4.2.3 Message Passing

Overview

ECS distributed computing consists of a variety of client and server applications running on different platforms. Clients send data to servers, which process the data and return the result to the client. This interaction can be classified into three categories: synchronous, asynchronous and deferred synchronous.

In synchronous mode, a client makes a request and passes control to the server, i.e., waits for an acknowledgment. The server services the request and returns the result back to the client, at which point the client gets back the control. The program execution on the client side is blocked until the server returns from the service. This is a blocking call and is referred as "synchronous".

In asynchronous mode, the client makes a non blocking request. Client processing can continue simultaneously with the server processing. Asynchronous message passing can be further subdivided into two parts: guaranteed and non-guaranteed. Guaranteed asynchronous message passing guarantees the delivery of the message to the receiver (client-to-server or server-to-client). The sender of a message can verify the delivery of the message through acknowledgments. In non-guaranteed asynchronous message passing, the sender should send the message to the receiver only if the receiver is active and listening. The receiver still may not receive the message due to any number of reasons, and this is not considered an error. FOS applications use this method to send real time telemetry data to ISTs asynchronously.

Deferred synchronous model is a superset of the asynchronous model. In deferred synchronous mode the client makes a call and receives an acknowledgment of the call, but no results; it also gets control back right away. Once the receiver completes doing any processing with the data, results are sent back to the client.

Table 4.2.3-1 describes the different types of message passing and when they are generally used.

CSS will implement synchronous, deferred synchronous, and guaranteed asynchronous message passing using Distributed Object Framework. Deferred synchronous communications involve some degree of application programmer involvement.

FOS applications like the Off-line Analysis Request process uses the message passing service to send analysis data to the Off-line Analysis process and to receive the results of such analysis.

Table 4.2.2.5-1. Security Object Responsibility Matrix

Class	Description
rgy_edit	This is command line interface used to create DCE accounts for principals, create keytab files for non-interactive principal (servers) to maintain its password, etc.
acl_edit	This is command line interface to manipulate (delete, insert, read, write, etc.) ACLs associated with objects.
DCEObj	This is a base class that stores information about the DCE interfaces implemented by a concrete manager class.
DCEUuid	This is the utility class that encapsulates the DCE data type uuid_t.
DCEInterface	This is a base class that encapsulates basic functionality for client objects.
DCEInterfaceMgr	This is a base class that encapsulates the functionality common to a DCE interface manager.
ESO	This is the Global Server Object. It manipulates manager objects and interacts with the DCE subsystems.
DCERefMon	This class provides an abstraction of a reference monitor that controls the client object's access to a manager object. By deriving from this abstract base class, various reference monitors that provide different security policies can be implemented.
DCEAcIMgr	This class registers a rdacl interface manager object with the global DCEServer(ESO) object. DefineAcIMgr macro is used to construct this class.
DCESecId	This is a utility class that encapsulates the sec_id_t data type.
DCEAcISchema	This is a class defining the ACL permission bits and print strings.
DCESchemaBitset	This class represents a set of permissions formatted according to a schema. Since most uses of ACL will use less than 32 bits of permissions, there is an efficient encoding of 32-bit ACLs.
EcsAcl	This class is used for accessing a DCE access control list. It maintains all the information about an ACL.
EcsAclDb	This is a class for ACL database. It defines the interface to the ACL database.
EcsModifyableAcl	This is a temporary copy of a EcsAcl that can be used for editing directly by the server or through an application-defined interface.
EcsAclStorageManager	This is a class that maintains a table of known ACL databases. It manages the ACL databases being used by a server, providing registration and search services for these databases.
EcsFilePassword	Provides an interface to the storage and access of password of principal (usually a server) to help the principal establish it's identity during login context.
EcsSecurity	Provides higher level functionality to authenticate principals accessing resources. These include create/update/delete ACLs, define permissions sets, persistence to the ACL database.

The FOS ECS Operations Control uses the message passing service to send schedule information to the ISTs.

SDPS process-intensive applications send the intermediate processing state/results to User Interface (UI) to display the results of the process done so far.

SDPS subscription service (between the science DataServer and the product generation) needs guaranteed asynchronous message passing.

CSS is providing two implementations for Message Passing. Both are designed to work with the OODCE-provided DCE-Pthread class which is used to start and control the execution of a thread.

The first implementation provides asynchronous and synchronous communications (byte streams only) with store and forward, and recovery - persistence. The concept of groups (one or more receivers) is included into this API.

The second implementation provides asynchronous and deferred synchronous communication. The CSS API for this design is not going to be as transparent to the user as the CORBA application will be, that is, some developer involvement will be required. With respect to Store and Forward, a thread can be created so that it can periodically try to send the message. The CSS API can handle any data types; recovery will not be supported.

4.2.3.1 Message Passing Implementation One

4.2.3.1.1 Overview

The first implementation will provide for asynchronous and synchronous message passing with store and forward, and recovery - persistence. The concept of "groups" is included in this API. A group is a symbolic name representing a number of receivers. Each receiver will be identified by a logical name associated with a UUID. This API will handle messages consisting of only byte streams.

4.2.3.1.2 Context

Message Passing is an infrastructure key mechanism and is used by ECS subsystems for synchronous, and asynchronous communication where a client needs control back immediately after invoking a remote procedure.

CSS will provide a set of classes (custom code) that will allow FOS and SCDO subsystems to achieve synchronous, and asynchronous communication. Security is available on this architecture.

Table 4.2.3-1. Message Passing Communication Types Defined

Type	Description
Synchronous	<p>Normally, this is how clients and servers interact using the Remote Procedure Call (RPC) mechanism. Both the client and server have to support the same interface. Servers are continuously listening for incoming requests. Each server, while supporting the same interface, can implement the interface differently. The client has the choice to bind to a server offering a particular implementation of the given interface.</p> <p>The type of arguments that can be used in this mechanism are all the system defined generic types plus user defined types. This is a blocking call, where the client has to wait until control returns back to it from the server.</p>
Non Guaranteed Asynchronous	<p>This is same as the synchronous message passing except that this is non-blocking, i.e. the client gets the control back immediately without waiting for the server processing to complete. In this mechanism, no result is returned. When a client invokes the request, the system invokes all the servers currently listening that support the interface. This is used primarily when the client wants to send some information to interested parties and doesn't care whether they receive it not. For example, when the CDS server comes up, it sends a message of its existence to all the (internal framework) directory agents that are listening.</p>
Guaranteed Asynchronous	<p>In guaranteed asynchronous mode, the sender specifies a list of receivers where he/she wants to send the message. A UUID is received back when the send request is issued. When a callback is returned indicating that the send operation completed, a UUID is passed back identifying which message was sent successfully or not. This is also a non blocking call, returning control to the sender immediately. The delivery of message is guaranteed in this mode. If a receiver is not listening, the message will be buffered and sent to the receiver at a latter time.</p> <p>The argument types that can be passed are not as general as in the above two cases, rather restricted to string type. This is used where large quantity of data needs to be passed, without any processing, e.g., the FOS Product and Scheduling element passing an instrument schedule to the SCFs.</p>
Deferred synchronous	<p>In deferred synchronous message, a sender sends a message to one receiver and gets back a UUID. This UUID is then used at a latter time to receive the result back from the receiver. This is used in process intensive applications, where the receiver takes some time to process the request. While the processing continues at the receiver, control is returned back to the sender immediately.</p> <p>As in guaranteed asynchronous message passing, the argument types are restricted to strings only.</p>

4.2.3.1.3 Directives and Guidelines

Directives:

The programmer must do the following in order to transfer data:

- Define a new class inheriting from the generic EcDcDSyncCom and must implement all the virtual functions defined in the parent class, PreInvoke, Invoke, and PostInvoke.
- Include the ClidDSyncCom.h header file.

- Create a client or a list of clients.
- Set the data pointer to some data that will be used in the overridden Invoke member function.
- Set the address pointer. The address pointer can be an IP, a port number, an object reference, a CDS name, etc.
- Optionally, set the thread policy attributes or just go with the defaults (EcDDcPri_min, and EcDDcFg)
- Optionally, set the number of retries in case of exceptions/errors. Defaults can be used.
- Optionally, set the time between retries in seconds. Defaults can be used.
- Call the method Send to transfer data.
- Call Done to find out whether the thread has finished execution successfully.
- Call Reset to set the flags to initial values and deallocate memory.
- Either re-send or terminate the session.

Guidelines:

This Message Passing implementation utilizes COTS such as OODCE and RogueWave. For information on OODCE please refer to the HP Object Oriented DCE C++ Class Library Programmer's Guide. For information on Rogue Wave please refer to SunSoft C++ 4.0.1 Tools.h Class Library.

It supports persistence. Intermediate queues are maintained, which collects all the messages to be sent, along with the receiver information and then sends them to the intended receivers. When a message is to be sent asynchronously, a pointer to it is placed in to an outgoing queue and control returns right away. The contents of the queue are saved to disk. If no file name is supplied when instantiating the control call, recovery/persistence is not provided.

Worker threads operate on the outgoing queue and will process simultaneous send operations. Each request that arrives is placed at the end of the outgoing queue. After adding the request to the queue, the boss thread will wake up a worker thread and this worker thread will perform the send operation. Once done, it will wait for the next request. If the message fails to be sent, it is placed back into the queue, the 'time_sent' field is updated, and, it is retried when it reaches the specified retry time. If it fails all the tries, the message is returned (callback is invoked with its UUID, destination, message, and send status).

Priorities are assigned to the spawned threads. Initially CSS is designed to support ten threads, with a configurable number of threads at each priority. A total of five priorities are defined. A higher priority thread tries to find a message from its corresponding queue and then sends it. In the absence of any messages in its queue, it looks for messages in the lower priority queues to send. Threads are assigned for each priority (even the lowest priority). This is done to ensure that lower priority messages are not starved.

In asynchronous mode, when the client sends the data to a set of receiving processes, a pointer to the data is kept in intermediate message queues and control comes back to the caller immediately. The message passing logic will locate the server who is continuously listening, and sends the data to the server.

The server can be running continuously to receive requests from the client. Alternatively, if the server is not running when the message is sent, the message passing logic can periodically try to send the message. The sender can specify the number of send-tries, and the time between tries. After sending the data, the sender will receive an acknowledgment indicating success/failure upon completion of the send operation.

In this mechanism, each application can receive as well as send messages to other applications. Each application maintains a group of outgoing queues in which outgoing messages are kept along with other information like destination. Each outgoing queue is associated with a priority. Messages belonging to a particular priority are kept in the corresponding outgoing queue. At startup, several threads are spawned whose primary purpose is to pop messages from the outgoing queues and send them to the proper destination. Each message is also associated with the number of tries, that is, how often it should be sent before declaring a failure (due to unavailability of the receiver or network failures) and the time period between each retry. The threads try to send the message the given number of times and calls an application supplied call back either after a successful send or after the determination of a failure to send the message.

A distributed object is defined with the functionality to transfer messages. After selecting a message from a queue, a thread creates a distributed client (a proxy object) and invokes the transfer method to transfer the message to the server object (the receiving application).

In order to receive messages, each application creates a receiving queue with a unique name. An application can create as many receiving queues as needed, each with unique name. This unique name is used by an application to send messages and it is kept in the Cell Directory Name Space (CDS) along with the binding information.

Each receiving queue is associated with a distributed server (a transfer object), which can receive messages from other applications. When a message is received by the server object, that message is kept in the queue associated with that object. The ordinary receiving queue provides two methods: read and read wait. These methods allow the application programmer to retrieve messages from the queue. The 'GetMessage' method performs a single read operation and returns a message from the incoming queue or Null (there are no messages in the queue) and a return status. The 'GetMessageWait' method performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call, or a default time period will be used.

Additionally a call back incoming queue is provided. This queue contains one thread which executes a programmer-defined call back every time a message is received. That is, a thread is awakened the moment a message arrives in the incoming queue, it invokes the virtual method 'handleMsg', and searches to see if any other messages arrived, if so, it executes another call back, else, it shuts down the thread until a new message is received. When a message is read, it is removed from the queue and from the disk.

Deferred synchronous model is a direct superset of the asynchronous model. While implementation one does not support deferred synchronous directly, it provides certain information, such as, the message UUID, so the application programmer can implement deferred synchronous messages in the application. Alternatively, the application programmer can use the CSS implementation two to provide deferred synchronous message passing as an infrastructure service.

Constraints:

- Each queue must have a maximum number of messages, a user-defined number.
- The messages must be of defined maximum length.
- Logical names ought to be unique, internally CDS/GDS will be used to store location information associated the receiver (logical name-UUID pair).
- The message or buffer should not be changed while the sending is in progress; otherwise, we need to copy the message.
- If deferred synchronous is needed, a UUID and the sender's logical name can be used to send a reply back by the receiving end.
- The Message or Buffer should not be changed while the sending is in progress. Otherwise, we need to copy the message.

4.2.3.1.5 Sample Application Programmer Interface

Sample #1

This scenario demonstrates how a process can be a sender of asynchronous messages and a receiver as well. It also demonstrates the usage of a receiver callback to receive a message (A process sends a message to another process and is able to receive as well).

Description

a. Setup an ordinary receiver session

Instantiate an object of the EcMpMsgPngCtrl type using a filename (for the outgoing queue and for the disk filename/persistence), and an application name (a full CDS path name + application name).

Using the message passing control object start listening for incoming messages (Initialize). Next, instantiate receiving objects, that is, create an instance of the incoming queue object (via the 'createReceiver' call) and set it to point to the queue the message passing control class is attached to it.

When a message comes in, it gets stored in to the queue associated with the object and a copy of it is stored on the file associated with the object.

In order retrieve a message two methods can be used: 1) GetMessageWait (block - there is wait) Performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call; 2) GetMessage (Non

Block - there is no wait) Performs a single read operation and returns a message from the Incoming Queue or a return status (“no more messages”).

b. Setup the sending session

A sending session is basically a point-to-point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side.

Create a callback object and implement/compile the acknowledgment virtual function. The object EcMpMsgCb has a virtual function named 'handleAck', this function is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constraints(number of tries). This function takes the following arguments: a flag indicating success/failure, the number of tries, the time between tries, the actual message, its length, a receiver address, and a destination address, and a message identifier. This same unique id is presented to the application programmer when sending a message.

Create a EcMpSessionList and populate it with the following information: 1) Associate the callback object just created, 2) Set the number of tries and the time between each retry, and 3) For each session give the receiver's (destination) logical name.

Prepare and send the message to the session list. At initialization time a number of threads were generated internally which periodically will get the messages from the outgoing queue and send them. The send call returns back a UUID so the sender that can be used by the callbacks. This UUID can also be used to get the reply back if there is one.

c. Setup a callback receiver session

The receiver will be notified everytime a message is received. A thread is awakened when a message is received in the queue. A callback is invoked at that point.

Instantiate an object of the EcMpMsgPsnCtrl type using a filename (for the outgoing queue and for the disk filename/persistence), and an application name (a full CDS path name + application name).

Using the message passing control object start listening for incoming messages. Next, instantiate receiving objects, that is, create an instance of the callback incoming queue object (via the createReceiverCb call) and set it to point to the queue the message passing control class is attached to it.

When a message comes in, it gets stored in to the queue associated with the object and a copy of it is stored on the file associated with the object. A 'call back' thread (just one thread) will be awakened every time a message arrives into the special receiver queue created by the call 'createReceiverCb'. A call back (virtual function) will be executed. It will check if there is another message and if so execute another callback. If there are no new messages it shutdowns the thread until a new message arrives in.

Step 1 Initialize, and Step 2 Setup Receiver Sessions

/*

The application programmer should include a set of include files given by CSS at the start of the application.

The EcMpMsgPsngCtrl object is the controller object, through which any number of receiver sessions can be created. Each receiver session is associated with a unique name (so other applications can send messages to this receiver) and a unique file. This file is used for persistence. When a message comes in, it is stored in to the queue associated with the object and a copy of it is stored on the file associated with this object.

Internally, this object also creates a sender queue. All outgoing messages are kept in this queue.

A number of threads are generated internally in the initialize call which periodically get messages from the outgoing queue and send them

*/

// Instantiate the controller object (initialize)

Extern EcMpMsgPsngCtrl* theMsgCtrlP;

theMsgCtrlP = new EcMpMsgPsngCtrl(fileName1), //to store outgoing messages
applName1); //CDS Name - applic. name

// Start listening

EcMpMsgPsngCtrl->Listen();

// Instantiate receiving objects.

EcMpQueueIn *rec1P;

EcMpQueueCbIn *rec2P;

// Create an ordinary receiver

// Open a disk file 'fileName2' for this receiver queue (for

// persistence purposes)

rec1P = theMsgCtrlP->CreateReceiver(receiverUniqueName,
fileName2);

```

// Instantiate the receiver callback object
EcMpMsgCb* msgRecCB_P = new EcMpMsgCb();

// Create a special receiver
// This receiver executes a virtual callback function
// the moment a message is received.
// Also, open a disk file 'fileName3' for this receiver queue
// (for persistence purposes)
rec2P = theMsgCtrlIP->CreateReceiverCb(receiverUniqueName,
                                     fileName3,
                                     msgRecCB_P);

```

Step 3 Setup the Sending Sessions

```
/*
```

A sending session is basically a point to point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when an asynchronous send is complete. This is done at the sender side.

Create a callback object and implement (compile) the acknowledgement virtual function. The object EcMpMsgCb has a virtual function named handleAck. This function is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constraints(number of tries). This function takes the following arguments: a flag indicating success/failure, number of tries, time between tries, the actual message, destination, a unique id. This same unique id is presented to the application programmer when sending a message.

```
*/
```

```

// Instantiate the sender callback object
EcMpMsgCb* msgSenderCB_P = new EcMpMsgCb();

```

```
/*
```

Create a sender session list and populate it with the following information:

1. Associate the callback object.
2. Set the number of tries and the time between each retry
3. For each session give the receiver (destination) logical name.

In case of a synchronous send method, the callback is not used. As such a null argument is a valid one in the constructor. Similarly number of retries and time between calls are applicable only for asynchronous sends.

```

*/

// Instantiate sessions lists and give information (callback, # of tries, destination names ...)
EcMpSenderSessionList *ssListP=new      EcMpSenderSessionList(msgSenderCB_P);
ssListP->SetTries (5,10);                // no of tries, and time between tries
ssListP->Join(LogicalName);              // logical name (stored internally in CDS with its
                                         // respective object UUID

ssListP->Join (anotherLogicalName);

```

Step 4 Send a message

```

/*

Send a Message: This returns (output argument) an UUID so the sender can use it in the
callbacks and the status of the call.

*/

// Send the message to the group (all in the group will receive it)
status<- ssListP->Send(WaitFlag,        // Wait (Synch) or Non-Wait (Asynch) Flag
                      msgP,            // A void pointer
                      msgLength,      // Msg length
                      senderAddr,     // Destination queue
                      recAddr,        // Client Queue. If NULL, no receiving at the client
                                         // will occur
                      priority,       // Priority 1/2/3/4/5
                      uuidFlag,       // UUID will be given or not
                      msgId);         // This is an output argument to identify the message
                                         // in the callback method

```

/*

In case of a synchronous call, control returns after the call is completed. In this case, the callbacks are not called. Callers main thread is used to send the message. In case of asynchronous calls, the message is put in a queue, which will be sent later by an internal thread.

*/

Step 5 Receiving a message

/*

a. Read - Block (there is wait): Performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call.

*/

```
rec1P->GetMessageWait(msg);    // wait and dequeue
```

/*

b. Read - Non Block (there is no wait)

Performs a single read operation and returns a message from the Incoming Queue or a return status ("no more messages")

*/

```
rec2P->GetMessage(msg);        // dequeue
```

```
// ... anything else you want to do ...
```

4.2.3.1.6 Object Model

Table 4.2.3.1.6-1 summarizes the first Message Passing Service classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.3.1.

Table 4.2.3.1.6-1 Message Passing Object Responsibility Matrix (1 of 2)

Class Name	Description
EcMpMsgPsnCtrl	<p>The EcMpMsgPsnCtrl object is the controller object, through which any number of receiver sessions can be created. Each receiver session is associated with a unique name (so other applications can send messages to this receiver) and an optional unique file. The file is used for persistence.</p> <p>When a message comes in, it is stored in the queue associated with the object and a copy of it is stored on the file associated with this object.</p> <p>Internally, this object also creates a sender queue. All outgoing messages are kept in this queue.</p> <p>A number of threads are generated internally in the initializing call which periodically get messages from the outgoing queue and send them.</p>
EcMpMsgCb	<p>This class will handle two types of callbacks:</p> <ol style="list-style-type: none"> 1. For ordinary receive messages: If an ordinary message is received, then handleMsg is invoked; 2. For acknowledgment of messages: If an acknowledgment is received, then handleAck is invoked. A sending session is basically a point to point session to send a message. Each sender session will have a logical name that is needed to contact the receiver. <p>A list of sending sessions are maintained in a given application. A sender session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side.</p> <p>A callback object is created and will implement the acknowledgment. The virtual function handleAck is called when a message is delivered to the destination or when the underlying mechanism failed to deliver it within the given constraints (number of tries).</p>
EcMpSessionList	<p>This is a container class whose element type is a logical name and will inherit from the RWTPtrSlist class.</p> <p>A session list contains a callback object which provides virtual functions to be called when a send is complete. This is done at the sender side.</p>
EcMpQueue	<p>This class will be the parent class for the following:</p> <ul style="list-style-type: none"> • EcMpQueueIn • EcMpQueueCbln • EcMpQueueOut <p>EcMpQueue inherits from the Rogue Wave library file RWPtrDlist.</p>
EcMpQueueCbln	<p>This queue will contain one thread which will execute a callback every time a message is received. The callback will be a virtual function call. This class defines a double linked list queue. It inherits from the Rogue Wave Library file, RWTPtrDlist.</p>
EcMpQueueIn	<p>This class will be used to queue the messages once they are received. It will provide a Read Wait call and a Read Non-Wait call. The Read Wait call performs a single read operation; however, this call waits until a message is read from the queue. Optionally a time to wait may be provided in the wait call, or a default time will be used. The Read Non-Wait performs a single read operation and returns a message from the Incoming Queue (or Null if there are no messages in the queue) and a return status. This class defines a double linked list queue. It inherits from the Rogue Wave Library file, RWTPtrDlist.</p>

Table 4.2.3.1.6-1 Message Passing Object Responsibility Matrix (2 of 2)

Class Name	Description
EcMpQueueOut	In case of a asynchronous calls, the message is put in a queue, which will be sent later by an internal thread. Controls returns immediately. Worker threads will process simultaneous send operations. Each request that arrives is placed at the end of the outgoing queue. After adding the request to the queue, the boss thread will wake up a worker thread and this worker thread will perform the send operation. The send operation will not remove the item from the queue yet. Once done, it will wait for the next request. There will be about five to ten working threads and one boss thread. If the message fails to be sent, then the 'noOfTries' gets decreased by one, and the 'lastTimeSent' gets updated to the current time (when the message came back after the send failed). The message will be retried once the 'lastTimeSent'+ 'timeBetweenTries' was reached until the noOfTries expired. If the message failed to be sent, the message will be returned (the callback 'handleAck' will be invoked). The EcMpQueueOut class defines a double linked list queue. It inherits from the Rogue Wave Library file, RWTPtrDlist.
EcMpTransferSrv	This class is the EcMpTransferSrv manager object. It responds to client's requests to transfer data which result in the enqueueing of the data.
EcMpTransferCli	Class EcMpTransferCli is a surrogate object for making requests to an EcMpTransfer manager object. An EcMpTransferCli object creates and holds a single instance of this class which it then binds to successive EcMpTransfer manager objects to carry the transfer operation.

4.2.3.2 Message Passing Implementation Two

4.2.3.2.1 Overview

Implementation two provides asynchronous and deferred synchronous communication. It is designed to work with the OODCE-provided DCE-Pthread class which is used to start and control the execution of a thread. The CSS API is not going to be as transparent to the user as the CORBA application will be, that is, some developer involvement is required. For example, with Store and Forward, the developer using the second implementation needs to create a thread to listen for a reply from the server. Reply sensing will be OTS with CORBA. In addition, recovery is not supported with the second implementation.

The CSS API spawns a thread to send a message. This thread invokes predetermined, application-programmer specializable, virtual functions. The API supports asynchronous communications.

The CSS API also supports deferred synchronous communication. The results produced by the execution of the thread can be retrieved using the GetResults member function.

The number of tries, as well as the time between each try, must be defined.

Thread scheduling attributes can be defined if the defaults are not desired. Scheduling policy controls the algorithm used to schedule threads. Scheduling priority controls the treatment of a given thread relative to other threads. The default scheduling policy is Foreground. All the threads with this policy will be scheduled on a round-robin basis regardless of their priority. Higher-priority threads will get better treatment, but all will get some time to run - to provide some fairness to low-priority threads. Foreground threads can still be locked out by higher-

priority thread types, such as, FIFO or straight Round Robin. The default scheduling priority will apply to the minimum symbol of the default scheduling policy.

Only one send call can be executed at a time for each method, that is, threads will not be launched concurrently. The developer can check the status of a thread by calling 'CallInProgress'. This function will return an integer, a '0' for currently in progress, a '1' for not currently in progress.

The programmer can call the method 'Done' in order to find out whether the operation has finished successfully or not.

The 'Reset' is used to deallocate any memory assigned to the results field and to reset the flags. After Reset, the developer can call Send again or just delete the client object and terminate.

4.2.3.2.2 Context

Message Passing is an infrastructure key mechanism and is used by ECS subsystems for deferred synchronous, and asynchronous communication where a client needs control back immediately after invoking a remote procedure.

This service utilizes COTS within the OODCE and DCE products. CSS provides one generic class with virtual functions which FOS and SDPS must inherit in order to achieve asynchronous and deferred synchronous communication.

4.2.3.2.3 Directives and Guidelines

The programmer needs to define a new class inheriting from the generic EcDcDSyncCom class and should implement all the virtual functions defined in the parent class, that is, PreInvoke, Invoke and PostInvoke. In PreInvoke, the user can do any initialization that is needed prior to the transfer, Invoke executes the actual transfer method. Once control returns back from Invoke, PostInvoke is called. This method can take care of the Notifications in the case of Asynchronous message passing.

The main idea behind this API is to allow the programmer to spawn a thread for listening purposes and then release it so that the main thread is available to perform other operations such as sending data.

The programmer must do the following in order to transfer data:

- Set a data pointer to some data that will be used in the overridden Invoke member **function**. It is represented as a void pointer, to allow for various types of data.
- Set the address pointer to some address that will be used in the overridden Invoke member function. It can be a port number, an IP, a binding, a CDS name, etc. The address is a void pointer. Since the developer, will be setting this field, he/she will know how to parse it and implement it in the Invoke call .
- Call the Send method. Internally this method will create a number of threads and invoke the 'EvalThread' function which will execute the PreInvoke, Invoke and PostInvoke virtual methods. If the send fails, the call will be retried as many times as defined by the

developer. Once the thread finished execution, it is terminated, but the client object will not be deleted. Results from the thread execution are returned, and 'reset' needs to be called to deallocate any memory priority set aside. The programmer can then either send some more data or terminate and delete the client object.

Assumptions:

- Suppose the client class inheriting EcDcDSyncCom had more than one method, then, for each method, an instance of the client class will be created, that is, for each one there will be one call, one thread executing at one time.

4.2.3.2.5 Sample Application Programmer Interface

Sample

This scenario demonstrates how a process can be a sender of deferred synchronous messages and at the same time operate as a server by listening for incoming requests (process sends a message to another process, and get a reply back).

Description

Step 1 First define a new class inheriting from the generic EcDcDSyncCom and implement all the virtual functions defined in the parent class, that is, PreInvoke, Invoke, and PostInvoke. Let the client class be CliDSyncCom.

For example define the class as follows,

```
Class CliDSyncCom : public EcDcDSyncCom {
    public:
        CliDSyncCom();           // Constructor
        virtual ~CliDSyncCom();  // Destructor
        // Implementation of pure virtual base class functions
        virtual EcTInt PreInvoke();
        virtual EcTInt Invoke();
        virtual EcTInt PostInvoke();
};
```

and override the virtual functions.

```
EcTInt CliDSyncCom :: PreInvoke()
{
    EcTInt ECSSStatus;
    // ... do some something before calling Invoke
```

```

        return ECSStatus;
    }
EcTInt CliDSyncCom :: Invoke()
{
    EcTInt ECSStatus;
    // ...
    // Suppose there was some class called 'Telemetry' with a method called 'transfer',
    // then, we set a pointer of type 'Telemetry', 'TelP', to point to '_addr', an IP
    // address, a port number, a binding, or anything of that type.
    if ( _data)
    {
        try // Set up try block for exception handling
        {
            Telemetry *TelP = (Telemetry *)_addr;
            // In the case of Deferred synchronous, we want to store the results
            // from the computation in '_results'
            _results = TelP->transfer(...);
        }
        // catch any DCE related errors and print out an informative string if any
        //occur
        catch (DCEErr& exc)
        {
            // ...
        }
    }
    return ECSstatus;
}

```

```

EcTInt CliDSyncCom :: PostInvoke()

```

```

    {
        EcTInt ECSSStatus;
        // ... do some something after calling Invoke. This method can take care of the
        // notifications in the case of Asynchronous Message Passing
        return ECSSStatus;
    }

```

Step 2

Include the following object definitions:

```
#include "CliDSyncCom.h" // contains the message passing object definitions
```

Step 3-12 In Main, include the client header definition file and spawn a thread for listening purposes, and start listening. Call `pthread_yield` to notify the Thread scheduler that the current thread will release the processor 'thread_listening'.

On the main thread create an instance of the `CliDSyncCom` object. Set the data pointer to some data that will be used in the overridden `Invoke` member function. Data can also be retrieved by means of the `GetData()` member function. Set the address pointer. It can be an IP, a port number, an object reference, a CDS name, etc. Optionally, set the thread scheduling attributes. If they are not set, the default values for each attribute will be used, that is, `EcDcPri_min`, and `EcDcFg`. Set the number of retries in case of exceptions/communication errors. Set the time between retries in seconds.

Call `Send`, a thread will get started and executed. The thread creation is done transparently, via this `Send` call. The developer does not have to deal with thread calls other than setting the priority or the scheduling policy if desired. Check whether the thread has finished execution successfully. Obtain the results that were a product from the thread execution.

Call `Reset` to set the flags to '0' and deallocate any memory used (i.e.: on results) . Delete the client object or reuse it.

```
EcTVoid main()
```

```
{
```

Step 3

```
// Create Clients. Instantiate a list of EcDcDSyncCom objects
```

```
EcDcDSyncComList *ClientList = new EcDcDSyncComList;
```

Step 4

// Create a client, an instance of the CliDSyncCom object

```
CliDSyncCom *client1 = new CliDSyncCom;
```

Step 5

// Set the data pointer to some data that will be used in the overridden Invoke member

// function. Data can also be retrieved by means of the GetData() member function.

```
client1->SetData( (EcTVoid*)params);
```

Step 6

// Set the address pointer. It can be an IP, a port number, an object reference,

// a CDS name, etc.

```
client1->SetAddr( (EcTVoid*)"/.../csmscell/RelA_Apps/TelemetryTransfer")
```

Step 7 - Optional, defaults can be used.

// Set the thread scheduling attributes. If they are not set, the default values for each

// attribute will be used, that is, EcDcPri_min, and EcDcFg.

```
EcEDcThreadPriority a_priority = EcDDcPri_low; // scheduling priority
```

```
client1->SetPriority( a_priority);
```

```
EcEDcThreadPolicy a_policy = EcDDcFifo; // scheduling policy
```

```
client1->SetPolicy( a_policy);
```

Step 8 - Optional , defaults can be used.

// Set the number of retries in case of exceptions/errors

```
client1->SetNoOfRetries( 5);
```

Step 9 - Optional , defaults can be used.

// Set the time between retries in seconds

```
client1->SetTimeBetweenRetries( 10);
```

Step 10

// Send a message in Asynchronous mode. The Invoke method is executed and returns

// control back to the caller. The PostInvoke method function is implemented such that,

// once the Invoke Call completes, PostInvoke notifies back the caller about it. There is no

// wait.

// When send is called , a thread will get started and executed. The thread creation is

```

// transparent to the developer
client1->Send();

Step 11

// Check whether the thread has finished execution successfully.
EcTInt job_status = Done();

Step 12

// Set the '_done' flag to '0' and deallocate '_results'.
client1->Reset();

Step 13

// Delete the client1 object or reuse it.

} // End of Main

```

4.2.3.2.6 Object Model

Table 4.2.3.2.6-1 summarizes the second Message Passing Service classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.3.2.

Table 4.2.3.2.6-1. Message Passing Object Responsibility Matrix

Class Name	Description
EcDcDSyncCom	<p>This class is used to achieve message passing using asynchronous and deferred synchronous communications. It is designed to work with OODCE-provided DCE-Pthread class which is used to start and control execution of a thread.</p> <p>The user is expected to override the virtual member functions of this class, which are, PreInvoke, Invoke, and PostInvoke, in order to perform whatever operations are needed.</p> <p>Provides the functionality to</p> <ul style="list-style-type: none"> - receive messages from sender - transmit them to the receivers - collect and maintain acknowledgment information from the receiver. - collect and maintain results from the receiver - pass the acknowledgment and results back to the sender.
EcDcDSyncComList	This class is used to send a message to a list of sessions in the group.

4.2.4 Multicast

4.2.4.1 Overview

Multicasting is a mechanism through which a single copy of data is transferred from a single point to several places. In the point-to-point communications, data is transferred from one application to another application. If an application needs to send the same data to several other applications, the same data is needed to be sent to each of the receiving applications. Multicasting allows a sending application to specify a multicast address and send one copy of the data to that address. This data is then distributed through the Multicast backbone to all the applications listening at that address. This reduces the network traffic and improves the performance.

4.2.4.2 Context

The multicast API(s) will be used by FOS for distributing three types of data:

1. Real-time telemetry
2. Events
3. Shared playback

FOS shall use other standard API(s) or protocols (such as TCP/UDP) for all other data types. Each of the three types is detailed below:

1. Real-time Telemetry

Real-time telemetry is sent from EDOS to the EOC. The data flow is unreliable and uses UDP/IP-multicast in conjunction with EDOS-provided multicast routers. The multicast IP groups will be predefined, and each group will reflect a different telemetry data stream. Thus, EOC hosts need only receive the data stream they are interested in and therefore do not require a mechanism to notify EDOS of group membership.

Each EOC Operational LAN host will receive telemetry directly from EDOS simply by listening to the appropriate pre-defined multicast address. The Multicast server will unicast the EDOS telemetry to the ISTs.

The multicast API(s) on the EOC hosts will be used for the receipt of EDOS multicast telemetry data. The API(s) will also be used by the Multicast server to forward telemetry to the ISTs. (The API will handle unicasting to each IST that is registered on the server.)

2. Events

Events are sent from any EOC multicast capable host in response to some significant change in status (such as a host connecting to a string or generating a shared playback). The event message will be sent to every machine on the network. (For instance, if a host on the EOC Operational LAN generates an event, the event message will be sent to all EOC hosts on the Operational LAN and all ISTs with sessions to Operational LAN hosts. The event message, in this case, will not be sent to the Support LAN hosts.)

The multicast API(s) will allow for generation and receipt of events, and will handle unicasting events to the ISTs.

3. Shared Playback

Shared playback consists of non-real-time telemetry being sent to one or more hosts at up to 12 times the real-time rate. The receiving group can consist of both ISTs and EOC hosts.

4.2.4.3 Implementation

ISS will provide FOS with an unreliable multicast service in the form of an API(s) providing C++ classes. The multicast service shall be implemented at the transport layer via UDP and at the network layer via IP multicasts or (where necessary) unicasts. The API(s) will allow multicasting between hosts located within either the EOC Operational or Support LAN (but not between the two). Communication to the ISTs will be via unicast, but the API(s) will shield this detail from the FOS applications, so that the application makes a call to send to single group and the API insures that the data is sent to all members (whether unicast or multicast) of the group. (Note that since this is a non-guaranteed delivery multicast service, neither the API(s) nor the underlying protocols insure data receipt; this must be handled by the FOS application.). This will be developed under the incremental track for Release B. As such, complete design of this section is not shown here. It is included here for information purposes only. This is a Release B service but code will be provided in Release A time-frame so that testing can be conducted by FOS.

4.2.5 Thread Service

4.2.5.1 Overview

A thread can be defined as a lightweight process. Threads actually exist within a process. Threads differ from the other Object Services in that threads do not involve networking. Threads are a local service that affect the operation of a single program on a single node. Threads provide an efficient and portable way to provide for asynchronous and concurrent processing, both of which are requirements of network software.

4.2.5.2 Context

All segments are expected to require the Thread Service for Release A. Due to the underlying complexity of threads, only a subset of their functionality is revealed to the other segments.

4.2.5.3 Directives and Guidelines

OODCE provides a set of classes to perform functions for the construction, monitoring, and control of threads for the POSIX threads model. The classes permit C++ developers to create new threads by using a class constructor, modifying the attributes of the executing thread and other threads, and constructing and using mutex locks and condition variables. Some defaults are provided in the implementation of these classes. For examples, mutex lock objects are automatically initialized when created. Developers do not need to make a separate call to initialize the locks.

Creating a Pthread does not necessarily start a thread. The Pthread is a proxy for communicating with a thread. Initially it holds desired attribute values. The Start operation, which may be invoked by the constructor or after the thread is constructed, actually calls pthread_create to start the thread. The Pthread object may be deleted either before or after the thread itself completes.

Implementing a multithreaded server does not require calling any Pthread routines. Simply calling the Listen() member function (in OODCE) will cause each invocation of a server operation to run as a distinct thread. Unless the server operation needs to create additional threads for some reason, there is no need to explicitly call the Pthread routines. Since each server operation executes as a result of an RPC, there is generally no need to synchronize their termination.

Implementing a multithreaded server/process, however, does require protecting against conflicts between different threads accessing the same data. Conflicts can occur because a thread can be time sliced at any time. Whenever a thread accesses data that can be modified by another thread, there is a potential for inconsistent behavior. A mutex is an object that multiple threads use to ensure the integrity of a shared resource that they access, most commonly shared data. For each piece of shared data, all threads accessing that data must use the same mutex; each thread locks the mutex before it accesses the shared data and unlocks the mutex when it is finished accessing that data.

You have control over the following attributes of a new thread: Scheduling policy attribute, Scheduling priority attribute, Inherit scheduling attribute and Stacksize attribute. The defaults are the following:

- ∞ Scheduling policy is SCHED_OTHER and SCHED_FG_NP (See below)
- ∞ Scheduling priority attribute is priority of the parent thread
- ∞ Inherit scheduling attribute is the new thread receives the priority and scheduling policy of the parent
- ∞ Stacksize attribute is machine dependent

All threads are timesliced. SCHED_OTHER and SCHED_FG_NP do the same thing; however, SCHED_FG_NP is simply more precise terminology. The FG stands for foreground and the NP stands for "new primitive". All threads running under the SCHED_OTHER and SCHED_FG_NP policy, regardless of priority, receive some scheduling. Therefore no thread is completely denied execution time. However, SCHED_OTHER and SCHED_FG_NP threads can be denied execution time by SCHED_FIFO or SCHED_RR threads.

A thread attributes object allows you to specify values for thread attributes other than the defaults when you create a thread with the pthread_create() routine. For information on the pthread attribute object refer to the OSF DCE Application Development Guide Core Components or the HP Object Oriented DCE C++ Class Library External Reference Specification.

4.2.5.4 Sample Application Programmer Interface

A few samples utilizing threads are shown below. For more information about the various methods refer to CDR documentation (Release A CSMS Communications Subsystem Design

Specification for the ECS Project, section 4.2.5), the HP Object Oriented DCE C++ Class Library External Reference Specification, and the HP Object Oriented DCE C++ Class Library Programmer's Guide.

Sample #1

Creating a new thread

Description

To create a new thread, construct a DCEPthread object and pass to the constructor the name of the routine to be executed and the argument to be passed to the routine. The following is an example of creating a checkpointing thread for a persistent database:

```
DCEPthread checkpoint_thread = DCEPthread(checkpoint, NULL);
```

checkpoint_thread is a DCEPthread object representing the thread of execution that is performing checkpointing. The DCEPthread object supports member function for changing the priority of the thread, checking the scheduling policy, setting its initial stacksize, and joining with the thread.

In this example, checkpoint is the name of the routine that is executed when the thread starts to run. Checkpoint() is passed a NULL argument in this example. When writing the checkpoint routine, declare the prototype as follows:

```
DCEPthreadResult checkpoint(DCEPthreadParam param);
```

The pthread attributes that had to be explicitly defined when using DCE directly are initialized to default value. To override the default values refer to the HP Object Oriented DCE C++ Class Library External Reference Specification or the man pages.

Sample #2

Creating a mutual exclusion lock

Description

To create a mutual exclusion lock, declare a DCEPthreadMutex object:

```
// protects a piece of data from being updated by concurrent threads
```

```
DCEPthreadMutex data_lock;
```

The Lock member function acquires the lock, and UnLock releases the lock:

```
data_lock.Lock();
```

```
data_lock.Unlock();
```

The mutex attributes that had to be explicitly defined when using DCE directly are initialized to default values. To override the default values refer to the HP Object Oriented DCE C++ Class Library External Reference Specification or the man pages.

4.2.5.5 Object Model

Table 4.2.5.3-1 summarizes the thread classes which are discussed in detail in the CDR documentation (Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.5), the HP Object Oriented DCE C++ Class Library External Reference Specification, and the HP Object Oriented DCE C++ Class Library Programmer's Guide.

Table 4.2.5.3-1. Thread Service Object Responsibility Matrix

Object	Responsibility
Pthread	Provides a lightweight process.
PthreadMutex	Synchronizes threads. Controls the access to data objects. Once a thread has locked a mutex, no other thread can use it or lock it until it has been unlocked.
PthreadCond	Coordinate the access to a mutex shared between many threads. Can advertise its' availability of mutex to many threads.
PthreadInterval	Used to specify synchronization intervals with a PthreadCond object.
PthreadTime	Used to specify synchronization intervals with a PthreadCond object.
ThisPthread	Provides the ability to reference the current running thread.

4.2.6 Time Service

4.2.6.1 Overview

The CSS Time Service will utilize the DCE (Distributed Computing Environment) DTS (Distributed Time Service) to keep system clocks in the ECS network approximately in sync by adjusting the time kept by the operating system at every node. Timestamps are used by many applications when recording event occurrences to a log. The implementation detail of the CSS Time Service and DCE DTS are invisible to the software developer.

The CSS Time Service will take advantage of the DCE DTS which has a Time Provider Interface (TPI). The TPI will allow an external time source to connect to the Time Service. A Time Provider provides access to standardized or government controlled time devices such as radios, satellites, or telephone lines. The servers with a Time Service query the Time Providers for the current time and can pass the standard Coordinated Universal Time (UTC) time values to a DTS server and propagate them through the network. The Time Providers are considered the most accurate source of time information.

The Distributed Time Service (DTS) synchronizes the system clock on each host by directly adjusting the time kept by the operating system. Under ordinary circumstances, this is done gradually so that there are no sudden jumps in the time. It is also done in such a way that the time never goes backward. If a system clock is too far ahead, it is slowed down until the time is correct by modifying the tick increment.

4.2.6.2 Context

All segments are expected to use the Time Service for Release A. The CSS Time Service will provide distributed time with millisecond resolution. Applications utilize the Time Service when they need to obtain the time in various formats. The Time Service provides APIs to perform these categories of functionality.

4.2.6.3 Directives and Guidelines

Since the CSS Time Service utilizes the DCE DTS API's the developer's process must be run in a DCE cell which contains DTS.

The developer must instantiate an `EcTiTimeService` object to use the CSS Time Service.

The CSS time service will not provide a method to set time but will provide methods to obtain the time in various formats.

A delta value must be placed in the namespace before instantiating an `EcTiTimeService` object if a delta is to be applied to the current time. Some applications may need to simulate the current time by applying a delta to the current time. The time class allows application developers to obtain the current time in various formats and optionally lets them apply a predetermined delta to those values. The delta value in the namespace should be a byte stream having the following format

[+|-]dd:hh:mm:ss where

[+|-] indicates plus or minus

dd indicates number of days

hh indicates hours

mm indicates minutes

ss indicates seconds

4.2.6.4 Sample Application Programmer Interface

A few samples of how to use some of the available EcTiTimeService methods are listed below. More methods for the EcTiTimeService class are described in further detail in the CDR documentation (Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.6).

Sample #1

Instantiate an EcTiTimeService object using a delta value

Description

If the developer wishes to use a simulated time, the developer needs to first create an entry in the namespace and set the value of the delta to be used to create a simulated time (refer to CSS Directory Services to create an entry in the namespace)

The developer may then instantiate the EcTiTimeService object by doing the following:

```
EcTiTimeService* ECSTimeP = new EcTiTimeService("NamespaceString");
```

Note: "NamespaceString" is the string for the name of the entry in the namespace where a delta value can be obtained.

Sample #2

Instantiate an EcTiTimeService object without using a delta value

Description

The developer may then instantiate the EcTiTimeService object by doing the following:

```
EcTiTimeService* EcTiTimeServiceP = new EcTiTimeService("\0");
```

Sample #3

Obtain current ASCII GMT time

Description

First the developer must instantiate an EcTiTimeService object.

```
EcTiTimeService* EcTiTimeServiceP = new EcTiTimeService("\0");
```

The developer should then call the GetAscGmtTime method as follows:

```

EcTChar*   TimeString;    // character string to receive the time
ECSSstatus = EcTiTimeServiceP --> GetAscGmtTime(TimeString);
Return value TimeString contains "(1995-05-16-13:23:31.215+00:00I-----)"

```

Sample #4

Obtain seconds and nanoseconds time values

Description

First the developer must instantiate an EcTiTimeService object.

```
EcTiTimeService* EcTiTimeServiceP = new EcTiTimeService("");
```

The developer should then call the GetSecNanoTime method as follows where seconds has been declared as unsigned long and nanoseconds has been declared as long:

```
ECStatus = EcTiTimeServiceP --> GetSecNanoTime(seconds, nanoseconds);
```

4.2.6.5 Object Model

Table 4.2.6.3-1 summarizes the time class which is discussed in detail in the CDR documentation (Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.6).

Table 4.2.6.3-1. Time Service Object Responsibility Matrix

Object	Responsibility
EcTiTimeService	Retrieving timestamp information Converting between binary timestamps and ASCII representations

4.2.7 LifeCycle Service (Initialization/Activation)

4.2.7.1 Overview

Distributed systems consists of applications distributed over several platforms connected across the network. The system is a collection of these applications. These applications collectively provide the necessary computational behavior to achieve the system objectives. Each application may consists of several objects. These objects provide some well defined functionality encapsulating the underlying state and implementation.

Managing a system involves managing individual applications. An operator may want to start a new application, shutdown/suspend a running application due to some anomalies. An application may not have to be active all the time to accept requests. For example, at peak times, one may want to run several instances of an application, while at other times, a few of the instances may be enough to service the incoming requests. In order to effectively use the CPU and memory, it is desired to control the applications as well as some objects residing in the application by starting them on demand.

LifeCycle services can be broadly classified into two categories: Application and Object level.

LifeCycle services for applications involve Startup, Shutdown, Suspend and Resume functionality on applications. This functionality lets the M&O manage server applications. MSS provides the application related LifeCycle functionality. CSS provides the internal APIs that are needed for the MSS to control the applications.

LifeCycle services for objects provide the application programmer with the functionality to create and delete server objects residing in different address spaces. These server objects listen continuously for incoming requests, process them, and send the results back to the caller.

4.2.7.2 Context

MSS will use this interface to request a server shutdown, startup, or stop/resume processing requests.

Application developers can control server objects through activation.

4.2.7.3 Directives and Guidelines

GSO is a Global Server Object defined in OODCE. The LifeCycle services, used to control applications, are implemented by redefining some of the functionality in the GSO object. The methods, the rationale along with a description of these changes are addressed in the following paragraphs.

- **Startup** - Starting up applications is done from the outside of an application. Management applications gather relevant data about an application and start applications from the Network Node Manager through the Management Agents residing on the hosts.

While startup doesn't need any functionality from within the application, shutdown, suspend and resume needs application cooperation. For example, before suspending an application, it may be needed to bring the application in a transient state (closing/saving files, finishing pending requests from other clients) for graceful shutdown.

Application related functionality is provided by the MSS, which in turn call CSS provided shutdown / suspend / resume services. Generally, in order for an object to service requests, it should be registered in several places together with the object-related information. Firstly, the object needs to be registered in a directory service which provides clients with partial information of how to reach the server object; the remaining of the information, that is, how to reach the server object, is kept into the end point mapper running on the host. Each application, when brought up, is assigned a port number on the host, where it can listen for incoming calls. The port number assigned is dynamic and can change every time an application is brought up. This information is kept in the endpoint mapper. Clients after getting the endpoint mapper information have to consult the endpoint mapper on the host to find the port number where the service is being offered. A client after acquiring all this information, makes a call to that port on a particular host. A control application (global server object) running on that port now receives the call and dispatches it to one of the objects residing in its address space. This is called runtime. In order for the GSO to dispatch the incoming call to the proper object,

the server object needs to be registered with the endpoint map. Shutdown, suspend, and resume control this information to provide / limit the ability of clients to reach server objects.

- **Suspend** - This method unregisters the information from the local runtime. The suspend provided by OODCE removes all the objects from the local runtime. In this case, the next resume call can not be a request coming from outside the applications as the application is not listening for incoming calls. The resume call must be originated from within the application. In order to listen to other control messages like resume, from the Management applications (Agent), the MsManager object's interface object must be up and must be in a listening state to receive incoming control messages. The Suspend method provided in the GSO should be rewritten to remove all the objects except the MsManager object from the local runtime. It also needs to remember the information about the suspended objects so it can register that information later when it receives a resume call. CSS provides this (custom developed) functionality by removing only the application objects from the local run time, and saves the information about the suspended objects with the Global Server Object.
- **Resume** - In order for the suspended objects to receive incoming requests, the information about these objects (suspended earlier) will be registered back with the local runtime. After the resume, the global server object listening for the entire server application can receive calls destined to the other objects in that application and then, dispatches them.
- **Shutdown** - Shutdown uses the OODCE provided shutdown method. It removes the object information from the local runtime, endpoint map. If the application binding information is registered in the CDS, Shutdown removes that information from the CDS. Removing the information from the CDS prevents new clients from obtaining information about the server objects. Clients with existing binding information can no longer reach the server object and an error status will be returned to the caller. In the process of shutting down an application, internally it calls the suspend first and then comes out of the listen loop and exit the application. Unlike regular suspend, shutdown needs to suspend all the server objects. So the suspend functionality provided will be to indicate that all the objects (including the MsManager object) should be suspended. The default provided shutdown function will be modified to call the custom developed suspend function before interrupting the listen loop.

The above mentioned LifeCycle operations are to be maintained by the application as a whole. In addition to that, functionality to control (create/delete) server objects is also needed. For example, some ECS applications like the DataServer may have to entertain several sessions (one per user) simultaneously to preserve the state information associated with each user. These session objects need to be created on demand. Creating a predetermined number of sessions objects will not only take-up unnecessary resources, but the number of session objects created may not be enough to receive requests from different users. Creating and deleting the server objects on demand can be done in two ways: Factories, and Activation. These are explained here in detail along with the pros and cons.

- **Factory:** A factory is a parent object which has the functionality to create child objects on requests. A session factory is to be running on the server side all the time which can take requests to create/delete sessions objects. Clients needing a separate session object must first bind to the factory object and make a request to create the session object. This session object is created exclusively for the requested client which can preserve the state information associated with that client. This reduces the complexity of maintaining state information in a threaded environment. After acquiring the binding information of a new session object, the client then interacts with that session directly to make requests.

Instead of explicitly binding to a factory object, and creating a session object in the client application, the client stub of a session object can be modified. A new client object for the session object is created inheriting from the IDL generated client stub. This new object when created, first creates an empty client object without binding to any particular session server. After constructing the client object, it binds to a factory server object (a server parent object which is capable of creating a session object) and requests the factory to create a session object. Obtains the binding information for the newly created session object and using this binding information, the session client rebinds to that newly created session server. This approach involves modifications (inheriting from the IDL generated client stubs, rather than changing them) to the client stubs, but provides a cleaner interface to the application programmer to create and connect to a session object. These modifications of the client stub are provided in CORBA at compile time. So migrating the existing applications to the CORBA paradigm will involve less breakage and makes the application programming logic much simpler.

- **Activation:** In this approach, each server object is associated with an activation object. While registering a server object, the activation object associated with that server object is also registered with the Global Server Object. The Activation object contains application programmer supplied information as to how create a server object. When a call is received by the GSO, it checks if an instance of that object exists in its address space. In the absence of an instance of that object, it calls the activation object to create an instance. The GSO then dispatches the call to the instance of that server object. The problem with this approach is that it only checks if there is an instance of a server object. It doesn't distinguish between instances of a server object. As such, it is not possible to create multiple instances of a server object within the same address space. This approach is useful for generic services (stateless servers), where one instance can receive and process all the incoming requests.

Note:

The Global Server Object (GSO) in OODCE needs to be specialized in order to modify suspend, resume, startup and shutdown.

4.2.7.4 Sample Application Programmer Interface

Please refer to the DOF Dynamic Model of the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.3.

4.2.7.5 Object Model

Table 4.2.7.5-1 summarizes the LifeCycle Service class which is discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.2.7.

Table 4.2.7.5-1. LifeCycle Service Object Responsibility Matrix

Class name	Description
DCEActivation	This is a pure virtual member function used to gain access to an implementation of activation functions. ActivateObject is called to activate the object associated with the UUID argument. The ActivateObject member function takes a DCEActivationResultT structure. Checks to see if the object state is stored in the file system, initializes the activation result and creates a new application manager object by passing the object UUID to the constructor.

4.3 Distributed Object Framework

4.3.1 Overview

Object Oriented applications consist of a number of interrelated objects. Each object is characterized by a set of attributes and methods. Each object has a clear interface that identifies the methods a user can invoke and get responses to. The object that requests information is called the requester and the object that provides a service is called the provider. Each provider object takes requests for operations that it has identified in the interface, performs the computations, and passes the results back to the requester. Object-oriented Application development consists of defining and instantiating the objects and passing messages (invoking methods) between the objects to achieve its objective.

In single address space applications, all objects reside in the same address space. In a distributed object framework, objects are distributed in multiple address spaces, spanning heterogeneous platforms. Objects can reside anywhere in the network, but the basic contract between an object and the users is the interface that the object provides and users can use. Objects can be spread across the network based on efficiency, availability of data. From the perspective of the requester of a service, the object location (location independence), invocation (invocation independence) should be the same no matter where the object is physically present.

Invoking methods amounts to passing messages between objects. If the objects are in different address spaces, then the messaging should be done via the network. The client/server paradigm supports this kind of communication. In this paradigm, one side of the session (client) is allowed to make requests, while the other side (server) may only make replies. Remote procedure call is one communication method that is used to implement the client/server paradigm.

The server provides a certain operation and is called a subroutine/function for the clients to use. In that sense, a normal program can be broken down into a number of subroutines and servers which implement the subroutines. Clients call these subroutines as if they are local. When a

client invokes one of these remotely implemented functions, program execution transfers from the client to the server where it is processed. Once the execution is done, the result is passed back to the caller. The client and the program execution flow will then be turned back to the client.

In order to achieve the above client/server interaction, there must be a standard interface definition language (IDL) to express the interface in a clear way. This is a pseudo language for which mappings should exist so that the interface expressed in IDL can be converted to high level languages like C and C++ using an IDL compiler. Once the interface is expressed in a standard language, any requester who wants to make an invocation can do so by adhering to the signatures present in the interface. The IDL should support standard types, obey some lexical rules and have a language syntax to express the interface in a crisp and unambiguous way and by preserving the semantics of the interface.

Since the data formats or internal representation of data may be different on different platforms, the RPC mechanism should provide a way to convert the data into a standard format so that both the receiver and the sender would interpret the data in the same way. The process of converting the data into a standard format is called marshaling and the process of converting the data from the standard format into a platform's internal format is called unmarshaling. This paradigm deals at the program function level and has no notion of objects.

An object framework is somewhat like the functional framework just described, but instead of differentiating at a function level, it differentiates at an object level. The object's behavior is captured in the interface definition language. The object's implementation is carried on remote hosts, which are responsible to execute procedures, update the object state and return the results. This paradigm makes use of inheritance while defining new interfaces by inheriting existing interfaces. Implementation inheritance may also be possible as long as the implementation of the super class exists within the scope of the current implementation. This paradigm also needs a standard interface definition language and provides the mechanism to marshal and unmarshal standard types. In this paradigm, an executing program (client) can instantiate an object at any host that provides the implementation of that object and query that object to do certain operations. In order to do that, two objects are created: one at the client and one at the server. The object created at the server is the real object that implements the behavior of the object. The object created at the client is called a surrogate object, whose main purpose is to marshal/unmarshal the arguments, make call to the real object, and get the results back to the calling program. From the client's perspective, the call is carried locally. The surrogate object does all the underlying work: locating the server that is offering the service, binding to it, setting security preferences, instantiating the server object if necessary through the use of life cycle services, and finally invoking the method. This is transparent to the client and is done through the use of IDL and the supporting framework.

The whole object framework consists of a set of core services with distinct functionality to make the development of the distributed applications easier. The core services are naming, security, threads, time, and RPC. In order to aid the application programmer, another layer of abstraction is provided. DOF interacts with the naming service to save and retrieve service locations. Similarly it interacts with the Security service to provide security. The other provided layer consists of four generic classes: ESO (inherits from DCEServer), DCEObj, DCEInterface and

DCEInterfaceMgr. Application programmers implementing the client server application need to develop three parts: an application client part which invokes the service, an application server part that implements the service, and an application server main (driver) part which actually creates and runs the application server as a separate process and provides all the functionality needed at the server side.

4.3.2 Context

FOS and SCDO clients use this framework to locate remote services, bind to those services and invoke methods provided by those servers. They pass the arguments to the server and get the results back from the server. Clients can set security preferences that they desire to have in communicating with the server.

FOS and SCDO servers use this framework to register the location of their services, set security preferences, receive incoming calls and redirect them to the appropriate implementation object, create/maintain Access Control Lists associated with methods defined in the interface.

4.3.3 Directives and Guidelines

The application client class inherits from the DCEInterface class and the application server class inherits from the DCEInterfaceMgr and the DCEObj to use the default functionality provided in the parent classes. Alternately, they can modify the inherited methods to achieve the needed behavior. There will be a global instance of the server class which the server driver uses. These four generic classes, along with the Interface Definition Language (IDL), C and C++ (limited) language bindings to the IDL provide all the functionality for the application programmer to develop client/server applications.

In DCE, the interfaces are defined using the DCE Interface Definition Language and are processed by a compiler called IDL, which generates system data structures and communication stubs for which generates system data structures and communication stubs for the client and server. OODCE uses the same IDL language, but an enhanced version of the compiler is used to process the interface specification. The compiler program is called idl++. The idl++ compiler generates the client and server stub and header files needed for a DCE RPC interface. idl++ also generates a number of C++ files that provide communication between OODCE clients and servers. Even though IDL++ generates C++ language stubs, it does not support interface inheritance and class attributes.

OODCE enforces the following:

- Each defined RPC must use explicit binding management. In explicit binding, the binding information (has the type 'handle_t') is passed as the first argument to remote procedures. DCE IDL also supports the implicit and automatic binding modes; however they must not be used in IDL files passed to the idl++ compiler. The functionality provided by implicit and automatic binding is supported at higher level in OODCE.
- An interface version number must be specified so that idl++ can generate class names that allow multiple versions of an interface to be used within an application without name clashes.

- Custom binding is an IDL feature not readily supported by OODCE. This can be supported at higher level and should not be specified in the IDL file.

The application programmer carries all the interaction with the underlying core services like Naming, Security, Threads, Time through the DOF for normal operations. Interfaces with these underlying core services are also provided and explained in earlier sections, through which the application programmer can tailor the application and to achieve finer grain control.

4.3.4 Sample Application Programmer Interface

Please refer to section 4.5 of this document for a descriptive Distributed Object Framework sample or to the HP Object Oriented DCE C++ Class Library Programmer's Guide for a variety set of samples.

Steps in writing a client/server application

The object framework provides the underlying infrastructure for remote object creation and method invocation. This section explains the process of developing a client/server application from the application developer's perspective. The development process can be divided into five basic phases.

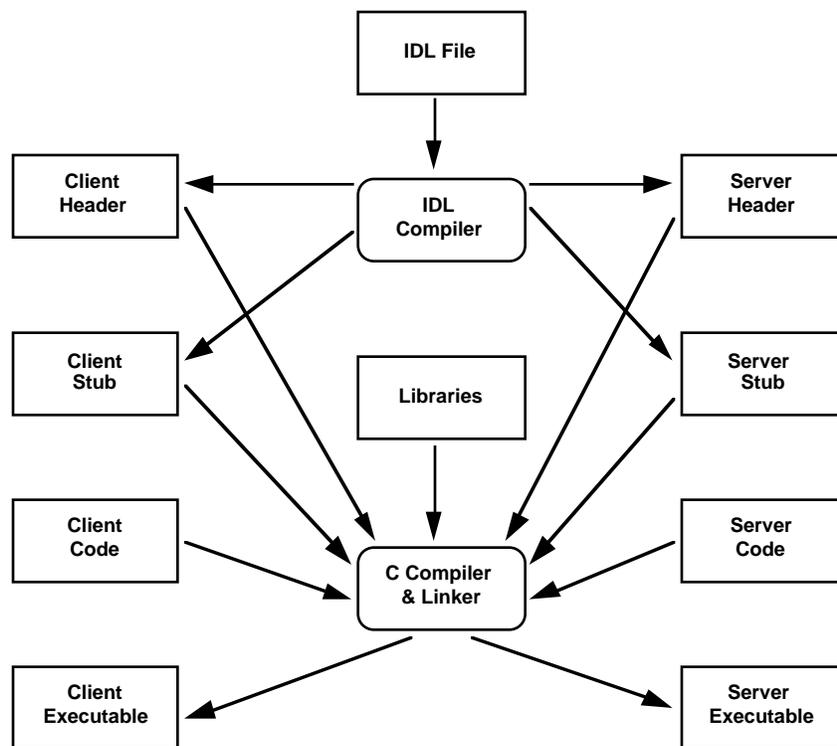


Figure 4.3.4-1. Client/Server Application Development Process

Phase 1 consists of defining the interface between the client, and the server in a pseudo interface definition language IDL. The interface expressed in IDL is then converted to stubs (written in a high-level language like C++) using a compiler. The stub files contain classes whose methods capture the behavior expressed in the IDL. The compiler generates the definition of two classes: one for the client and one for the server. The class generated for the client is called a proxy class. When an operation of this class is invoked, this operation in turn calls the corresponding operation on the server class that actually implements the behavior of the operation (method). The server side class acts as a manager class that takes calls and invokes the proper implementation of a method. Client side code is developed by imbedding the client class and making calls to it. Server class is embedded in the server code, which provides the actual implementation of the methods, along with other private functions.

Phase 2 consists of setting up the server to listen for incoming requests. On the server side, the same interface can have different implementations. Each one is called a manager. Setting up the server involves the following steps:

- Identify all the different implementations the server is going to support.
- Implement methods for each implementation (provide code for the methods)
- Identify the protocols to be used in servicing the requests.
- Specify the maximum number of threads the service can run in order to execute the user specified services concurrently.
- Register the authentication information.
- Establish the server identity (Reset the principal identity to the server principal name from the principal that is actually running the service).
- Register (export) the binding information for each combination of the interface name (UUID), protocol, and implementation in a central database.
- Obtain endpoints on the host for each such combination of the service and register them in the endpoint mapper on the host.
- Start a separate thread and go into a listen loop and wait for incoming requests.
- Wait for the thread to finish, or wait for a shutdown message or user interrupt (kill signal) and cleanup the central database by removing these services binding information and exit.

Phase 3 consists of a client binding to a service and invoking the service. The client obtains the binding information (binding handle) from the central database by specifying a protocol and a combination of a unique service name, a unique implementation type, a unique object name. The client then specifies the security options and invokes the services using the binding handle. [The arguments to the calls are marshaled and sent to the server.] This is a blocking call and as such, the client waits for the results from the server.

Phase 4 consists of a server receiving a request and processing it. When the server receives a request, [arguments passed are unmarshalled] it first authenticates the client and then wakes up the appropriate manager to process the request. The manager in turn, obtains the users privileges and compares them with the access control lists associated with the service to decide whether the

user can perform the operation or not and can return if the check fails. Otherwise, the manager calls the appropriate method to process the request, [marshals the result] and returns the result back to the caller and then goes into the wait loop waiting for other incoming requests.

In phase 5, the client receives the result back from the server [and unmarshals it] and continue execution.

4.3.5 Object Model

Table 4.3.5-1 summarizes the DOF classes which are discussed in detail in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.3.

Table 4.3.5-1. DOF Object Responsibility Matrix

Class name	Description
ESO	This class inherits from DCEServer and deals with the management of the objects that implement the application interfaces. This class provides the functionality to interact with the naming and security services. There will only be one instance of this class in a process. An instance of this will be created and bound to a global variable called theServer. Server main application uses this class to manage the objects.
DCEInterface	This class provides the basic functionality required at the client side. It includes locating a service, binding and accessing the remote object managed by that server. Application client class inherits from this class and as such inherits its behavior.
DCEInterfaceMgr	The DCEInterfaceMgr class is the base abstract class from which all generated server side classes are derived. It encapsulates object and type information as well as the endpoint vector (EPV) that is called by the RPC sub-system when an incoming remote procedure is received.
DCEObj	This class provides the concept of the Generic object that is a logical entity that can have multiple interfaces. This class is used to collect related interfaces together to form a DCE object. Object of this class can be registered with a Server object.

4.3.6 DOF Frequently Asked Questions

1) How a proxy object can reach a particular server object instance ? How to create a proxy using the object UUID ?

There may be several instances of an object. They all share the same interface UUID, but the object UUID will be different. The proxy object will need to use the object UUID to identify the specific server (the interface UUID is implicitly used in the call).

For example, if there is one DataServer running at each DAAC site and they all share the same interface id, in order to locate the one in Alaska, the caller needs to use the appropriate object id.

2) When session objects (server objects) should be deleted ?

When an application programmer creates a session object on the server side, it is the responsibility of the calling application (client) to delete these objects when done.

If the lifetime of these objects exceeds the application lifetime, then the application must be coded such that it remembers that information when creating server (manager) objects, and when the application comes up next time, it can either use the existing server objects or destroy them.

3) What is meant by combining interfaces ?

OODCE IDL doesn't support interface inheritance. If the programmer wants to combine several classes into one (a new class inheriting from all the classes) then it must be done outside of the IDL specification. First the programmer generates the IDL files and generates the server and client stubs separately. Then a new C++ class can be created which inherits from these IDL generated classes. This is done at both the client and the server side. The new object, which inherits from the IDL generated stubs, creates individual interface objects.

When the user creates one of these composite proxy (client) classes, it binds to the respective server objects. Calls made to the client objects relay the calls like before to the appropriate server objects. In order to relate the server objects, it may be necessary to maintain some information common to all of them.

4) How to set security preferences ?

Please refer to the Security Section, 4.2.2.

5) How to create and attach refmon objects to server objects ?

Please refer to the Security Section, 4.2.2.

6) How to check authorization?

Please refer to the Security Section, 4.2.2.

7) How to detect when a server has died and how to take appropriate action (communicating OODCE Exceptions from Server to Client) ?

By default, the server uses exceptions to communicate errors to the client. There are two ways to find server failures as described below:

- a. Propagate exceptions raised by the OODCE library directly to the client program without translation. This happens automatically if the server does not catch the exception. In this case the client must recognize and handle OODCE errors directly.
- b. The server program catches an OODCE library exception and translates it into an application-specific exception that the client program is more likely to know how to handle.

In general the second approach is recommended, although there might be situations where the first approach is more desirable.

There are a few exceptions converting errors to exceptions. When a server dies, the application can not convert the status to an exception and propagate it. In this case, the only way for the client to find out is to explicitly receive the status as an argument of the call and check it for server failure errors.

A call can have an additional communication status parameter. A client makes a call and waits for the server to finish processing. When the server dies at this point, control returns back to the client immediately with the failed communication status. The client can then check the communication status and decide what it should do. The status parameter can be added automatically to every call in the stub by generating another type of file called the Attribute Configuration File (ACF), which needs to be generated by the application programmer. For example, consider the 'sleeper.acf' attribute configuration file:

```
interface sleeper
{
    Sleep([comm_status,fault_status] st);
```

}

The Sleep call uses 'comm_status' and 'fault status' attributes. This controls the behavior of the stub code. 'comm_status' indicates that communication problems should be returned to the RPC caller as a DCE error in the 'st' parameter; 'fault status' causes fault in the server (such as data segmentation fault' in the manager routine) to be sent back as a DCE error in the st parameter. The 'st' parameter must be defined as an output parameter in the 'idl' file.

Another way to detect whether a server died, is when a server application creates a server object in its address space, to constantly watch when the last interaction with that server object took place. After a certain predetermined period of inactivity, the factory object that originally created that server object can delete that object.

Once, the communication with a server results in a failure the client object can be instructed to attempt a rebind. The method 'SetRebind' with the parameter 'attemp_rebind' will allow the client object to attempt to rebind and replay the failed operation. The default rebind algorithm is to first reset the endpoint on the binding handle used. If this fails, then an attempt is made to obtain a new binding (which could mean going to a totally different server). Rebind is attempted once if the rebind of the replaying of the failed operation presents another failure and an exception is passed to the client.

8) How to detect when a client has died and how to take appropriate action ?

When a server receives a call, it processes the request, and returns the result back. If the client dies in between, at the time of returning the result back, the server would find that the client is not there anymore.

The context handle provides this ability. When a context handle is created and passed to the client, the DCE runtime library keeps track of the connection between client and server; this may be done in the network code as in the case of TCP, or by DCE-specific ping messages if a connectionless protocol is used. When the client dies, the server is notified and executes a "rundown" function to clean up its data structures.

If ECS applications maintain their own queues, then the client is really not waiting for the server to process the call. The duration of the call is going to be very small, and the server is going to find that the client connection is broken.

If the processing involves some time, and the client is waiting for the return value during this processing, then the server needs to know the failure of a client as soon as possible (to stop further processing); this can be handled by passing client specific information. With DCE RPC, the solution is to associate the context with a 'context handle'. When the first call is made, the server allocates some kind of data structure that describes the current state of processing associated with that particular request and sets the context handle to point to it. Then the call returns, the context handle is returned as an output. The client then provides the handle on all subsequent calls. This allows the server to find the appropriate context. In IDL, context handles are declared by using a void pointer and the 'context_handle' attribute: typedef [context_handle] void *context_hdl_t;

It is important to know that context handles are associated by the runtime with specific servers. In DCE terminology, context handles carry binding information. A server that uses context_handle should provide a context rundown routine (a user defined function). This routine will be called by the RPC runtime if the connection to the client is broken (asynchronous notification during server processing). The purpose of the routine is to clear any resources associated with the context. Otherwise, the server would accumulate obsolete context information and perhaps eventually run out of memory or some other resource. The context rundown routine is identified by its name, which must consist of the context handle type name followed by '_rundown'. The rundown routine has one parameter which is the context handle to run down. Naturally the type of parameter is the declared context handle type. If the server uses more than one type of context handles, it should provide a rundown routine for each one.

OODCE produces the client/server stubs in a fixed way without this context handle as one of its default arguments in the method. CSS can not automatically include this argument in each of the application provided methods, as the stubs are generated from the compiler. Any change in the way the stubs are produced, should involve modifications to the compiler. The application programmer needs to modify manually the stubs to include a context handle, and define a new rundown method which will be called upon breaking a connection with a client.

9) How to pass an object as an argument in an RPC call ?

The communications infrastructure selected for the development of client/server applications is OODCE. OODCE uses an RPC mechanism to invoke functions on objects residing in different address spaces. Object passing, on the other hand, involves converting the object into a common network representation, passing it to the other object in a different address space, where the original representation is recovered (unmarshalled), and recreating the object at the receiving end. In order to do that, the IDL compiler should have the knowledge of the structure of the object that is being passed. OODCE doesn't support object passing in the versions to be used for release A, but does support an interim alternative to support this type of functionality.

In a distributed environment, desired functionality can be achieved without passing objects between applications, as objects are reachable (as such methods in them are invocable) from other applications. Some times, efficiency demands moving an object to a different application. In order to achieve object passing, the application programmer can define operator methods to convert an object into a common network format and from the common network format back to an object. The definitions of remote procedure call should be changed to change the type of the argument from an object to a byte stream. Then when a remote call is made with the actual object as an argument, the object is then cast (implicit conversion) to a byte stream and sent to the receiving object. The receiving object after receiving this byte stream, converts it into the proper object. DCE 1.2 is currently planning to support XIDL compiler which supports object passing in RPC calls. Until then, it is the responsibility of the application programmer to provide methods to convert the object into / from a byte stream.

In order to provide the functionality to send an object or any of its descendants, the receiving application should be able to distinguish which object it is receiving. For example, if CERES and ASTER are two objects inheriting from a generic DATAGRANULE object. The receiving object's call signature will contain a type DATAGRANULE. Now in order to distinguish the

actual object that it has received, the receiving application needs to have some indication of the type of actual object. This can be achieved by the sending application prepending a token like CERES or ASTER to the byte stream before sending it to the receiver. The receiver after examining the first token in the byte stream, constructs the appropriate object and recover its state information from the remaining byte stream. Another way to achieve this is by inheriting all the objects from the RogueWave class RWCollectable class. This class provides the necessary token information so that recovery of the object is easier for the application programmer.

10) How to deal when operating in several environments simultaneously ?

When Release A is in operational mode, we will need to test the next release simultaneously. While it is possible to carry on the testing in a separate environment, it is very expensive to maintain two sets of hardware environments. Running both the operational and testing in the same environment poses some problems. This section addresses some of them and explains how to deal with them.

Each server application consists of one or more server objects. Server objects (all instances of a given server object) are identified by a unique interface identifier. This identifier is the same for all the instances of a particular server object. For example, all the DataServer Server objects have the same identifier. In ECS we may have more than one DataServer object. These individual instances are identified by another unique identifier (object UUID) to distinguish among the instances of DataServer. Together, these two identifiers are enough to identify an instance of a server object. Server applications (NOTE: server applications and not server objects) also can be denoted by logical names. When a server comes up, the binding information that is needed for clients to reach them are saved in a central namespace with the above information. The identifiers and the logical names act as database keys to retrieve the binding information from the database.

Each interface (class) is also denoted by a version number consisting of two numbers: a major version and a minor version. Like operating system version numbers, the version number provides information to clients about compatibility. A change in the major version indicates that the previous version is not compatible with the new version of the server object. As such client applications compiled with the previous definition of the object can not bind to the newer version and invoke calls. On the other hand, an increase in the minor version indicates upward compatibility with previous versions of the server objects. A client compiler with a lower minor version can still reach a server object whose minor version is greater than the one in the client application. The version number is part of the IDL specification and is included in the stubs (client and server) generated by the IDL compiler.

A client can bind to a server object in various ways depending on the arguments that the client (proxy) object takes at construction time as explained in the previous scenarios:

- a. With no arguments - uses the interface identifier of the server object and binds to the first server object that it can find in the namespace
- b. With the logical name (CDS name) - finds the object associated with the logical name and binds to that server. The logical name is for the server application which can contain

more than one instances of a server object. In this case, the proxy binds to any of the server objects (of a particular interface/class) present in that application.

- c. With an object identifier (object reference) - finds an object whose interface and object identifier matches with the given information. This is the only way to find and bind to a particular instance of a server object.
- d. With the host address and the protocol sequence. The client object will go to the host porter and try to find a server program that implements its object. Once a server program has been located (using the Endpoint Map on porter) an object associated with the server is chosen to handle the client request.

The objective we want to achieve is that operational clients should bind to operational server objects, and testing clients should reach only the testing server objects. In order to achieve this, all the application development should follow certain directives as explained here.

Directive #1. Use compiler pragmas OPS and TEST to compile applications.

Directive #2. Where ever a resource (version number or a unique identifier or a file) is used, with the use of the compiler pragmas, generate two sets of them and use them.

For example: ifdef OPS

```
    uuid1
else
    uuid2
endif
```

11) How to get binding information ?

A client must locate a server compatible with it, this is called binding. The runtime routines that have to do with the use of the DCE Directory Service for the purpose of storing and retrieving binding information are collectively called the Name Service Interface (NSI).

A partially bound binding handle refers to a protocol sequence and server host (but not to an endpoint). The binding to a server cannot complete until an endpoint is found. A binding handle that is partially bound corresponds to a server system but not a server process on that system. When a partially bound binding handle is passed to the RPC runtime library, an endpoint is automatically obtained for you from the interface or the endpoint map on the server's system. A a binding handle that has an endpoint as part of its binding information is called a fully bound binding handle.

You can use the DCEInterface class (GetBinding) to return the current binding information (rpc_binding_handle_t). Also you can use DCEObjectReference Class (DCEGetBindings) and pass an object reference to obtain the RPC binding handle back.

The NSIObject base class defines the common behaviors of all objects in the directory, regardless of whether they are server entries, groups, profiles, or other types of objects. An

NSIObject can access bindings associated with the entry in the directory namespace, possible recursively if the object actually refers to a group or profile. A method is provided to retrieve a vector of binding handles either as an `rpc_binding_vector_t` or a `BindingVector` class.

Each NSIObject has an associated confidence level, indicating whether requests to the object can use cached data. A confidence of `LOW` indicates that the NSIObject or any derived class can do internal class caching; a confidence of `MED` indicates that the class itself cannot cache, but the CDS clerk cache may be used; a confidence of `HIGH` means do not use even the CDS cache, instead all request should go directly to the directory. The default provided by NSIObject is `LOW` to allow the class library to cache for improved performance.

In summary:

- low - get info from cache first
- medium - get info from the CDS replica first
- high - always get info from the master CDS.

12) How to reclaim unused server sessions objects ?

Need to record time of last access. server application periodically polls and deletes the objects that are older than a certain period.

13) Do two instances of the same server object share ACL files ?

Two instances of the same server object do not share ACL files unless they are in the same process. The problem is maintaining consistency between the application and the `acl_edit`. Please refer to the Security Section, 4.2.2.

14) What happens if there is a version mismatch ?

The version consists of two numbers separated by a dot. The first is the major version and the second is the minor version. Like the `UUID`, the version is used in the binding process. A client and a server are compatible only if the major versions are the same. In addition, the minor version of the server must be newer (higher) or equal to that of the client. If the minor version is higher, it indicates a compatible upgrade to the interface (interfaces that share the same major number and have a higher minor number are fully compatible with interfaces that have a lower minor number). Because the version of an interface is so significant, the DCE interface names include both a `UUID` and its version.

The logic behind this is that in some sense the server owns the interface. It implements the service and determines how it works. The client must take it or leave it. Thus, the server is never modified to respond to changes in the client, but the client may need to be modified if incompatible changes are made to the server. So when modifying the server, the rule is: if the changes are upwardly compatible (old clients will still work), increment the minor version only. If the changes are not upwardly compatible, increment the major number and set the minor version to zero.

FAQ #10 also describes the version attribute.

15) What is meant by an interface in DCE RPC?

An interface is a set of remote procedure call operations and associated data. Every interface contains one or more operations. An operation is an actual remote procedure. Each operation may have input and output parameters associated with it, just like any procedure call.

16) Can a DCE client import multiple interfaces?

Yes. A client can use as many different services as it needs.

17) Can a DCE client connect to multiple servers?

Yes. A client can connect to multiple servers providing different services, and/or multiple servers providing the same service. To use multiple servers with the same interface, the client must obtain a binding handle for each server and use explicit handles in the RPC.

18) Can a DCE server export multiple interfaces?

Yes. A server can provide service on multiple interfaces simultaneously. A common example is a server which exports an application interface and a management interface.

19) Can a process be both a server and a client?

Yes. There are two scenarios.

- A program might act as a server for interface A, and also as a client for interface B. This is easy. The program merely imports interface B like a normal client and exports interface A like a normal server.
- A program might want to provide a service, and also act as a client to other servers that provide the same service. In this case, the programmer must expend more effort. The problem is that the names of the server-side functions (manager routines) clash with the names of the client stubs. The solution is to manually build an endpoint vector for the server, and use different names for the manager routines. For details on using endpoint vectors, see the Lockhart book.

Note that most server programs also act as clients, since they usually access the endpoint mapper (rpcd), and the security service; these actions use RPCs, though it may not be obvious in the code.

20) How do I perform asynchronous RPC?

DCE-RPC is synchronous. The way to make an asynchronous call is to create a thread for each RPC call. You should be able to have dozens, if not hundreds, of threads with no problem.

21) How can a server keep track of multiple clients?

For example, to know what information has already been provided to a client, and thus vary subsequent responses.

The DCE RPC mechanism includes a "context handle" which can be created by a server and returned to a client. The handle is used on subsequent RPCs to identify the client.

22) How efficient is DCE RPC?

Performance testing at several user organizations has shown that DCE RPC performance is similar to other RPC implementations when doing the same things. The throughput and response times for a series of remote procedure calls is similar.

The use of features in DCE not present in other implementations may consume additional time and resources. For example, name-based binding may require additional time, depending on the number of directories traversed. Using the packet integrity and packet privacy features of the security service can increase processing times as a linear function of message sizes.

There are three papers providing preliminary performance data published in:

DCE--The OSF Distributed Computing Environment, Lecture Notes in Computer Science #731, Springer-Verlag

23) The DCE threads uses draft 4 of the standard, but DCE threads standard has moved beyond draft 4. Will DCE change to the most recent standard?

It is hard to predict exactly what will happen. But OSF prefers to follow standards rather than invent them. Once the threads document is approved as a standard, it would be obvious for the DCE to migrate to it.

24) Is DCE IDL the same as all the other IDL's in the world?

No.

IDL stands for "Interface Definition Language," and the idea of using a special language to define the interface between entities is not unique to DCE. In particular, CORBA's IDL is used for the same purpose as DCE's, but the two languages are not identical; see Q26 for more information. There are other Interface Definition Languages as well. IDL also stands for "Interactive Data Language", which is a completely unrelated product.

When asking or answering a question about IDL, one should be careful about specifying which IDL is involved.

25) Can I move idl-compiled stubs from one platform to another and rebuild the object files locally?

No. You must run the IDL compiler separately on each platform.

The IDL compiler builds the client and server stubs to handle network communication and data marshalling, which are platform-specific activities. Therefore the stub code is not portable and must be re-created on each platform.

Likewise, while the task of the stub does not change, the set of service routines called from the stub may be changed by the vendor for any given platform. Therefore stubs for the same RPC may look very different on different platforms.

26) Does DCE Security interoperate with other Kerberos systems?

Basically, no, or maybe yes, depending on what you want to do. To use authenticated DCE services, you must have credentials from the DCE security service; vanilla Kerberos v5 tickets aren't sufficient. But then, to use DCE services you must be using DCE RPC, so this is not really a problem.

Going the other way, it is expected that a DCE security server can issue tickets that can be used by vanilla Kerberos applications. The OSF was wary of promising this until the Kerberos v5 specs were published, but now that the Kerberos RFC has been published, OSF anticipates guaranteeing interoperability sometime "soon".

In a little more detail, the way to think about this is as follows:

Kerberos offers 2 services (Authentication Service, Ticket Granting Service) over 1 communication mechanism (UDP port 88). DCE security offers 3 services (AS, TGS, Privilege Service) over 2 communication mechanisms (UDP port 88, RPC). Where Kerberos and DCE security intersect (AS, TGS over UDP port 88), the services are identical.

27) Can I use DCE from C++?

Yes. First of all, since you can call C functions from C++ you can access all the DCE services from a C++ program. But that will not give you the benefits of C++. OODCE provides a C++ interface to DCE.

28) Can I write an application that uses DCE and X11/Motif?

Yes, but there are several serious pitfalls.

The X11/Xt/Motif libraries may not be thread-safe. For example, suppose one thread calls a function in Xt, which calls a non-thread-safe malloc(), which then gets preempted. The next thread may call a thread safe malloc() that comes with DCE. When control returns to the first malloc(), any assumptions about the state of the heap are invalid.

Also, Motif/Xt/Xlib are not currently reentrant wrt/themselves. You can't have multiple threads concurrently manipulating any Motif/Xt/Xlib global state. Fortunately this issue is under your control when designing the application. X11R6 includes a thread-safe version of Xlib, but it will be a while yet before the vendors are all delivering thread-safe Motif.

A related issue is that XtAppMainLoop() waits in a select() for activity, coupled with the fact that DCE also waits in a select() for activity. Unless the two are select()s are cooperating, one or the other will be starved. This is a platform-specific issue, you should check with your DCE vendor for full details. If it is a problem in your environment, the standard solution is to encapsulate the GUI in one process, the DCE client code in another process, and connect them with a simple IPC such as a Unix pipe.

29) Is DCE an official standard?

OSF calls the specification an Application Environment Specification, or AES. The AES documents both the software programming interfaces and also the communications protocols employed by DCE. Thus it would be possible, in theory, for someone to build a compatible implementation without using the code from OSF.

The AES for RPC, Time, and Directory services have been accepted as standards by X/Open. The AES for Security is currently undergoing review.

DCE Threads follow the Posix Threads draft standard 1003.4a draft 4. DCE Access Control Lists (ACLs) are based on POSIX.6 Draft 12. The Distributed Time Service (DTS) uses time formats defined by international standards and in POSIX.4. The Global Directory Service (GDS) complies with the X.500 international standard. (Although DCE complies with the 1988 version of X.500, not the 1992 version.)

ISO is considering an RPC standard based on the X/Open document.

DCE's status as a de facto standard is even stronger. Almost every major hardware and software vendor has committed to providing DCE on its platform. These vendors include not only OSF stalwarts such as IBM, DEC and HP, but also other key vendors such as Novell, Inc. See Q 6 for a list of DCE vendors. In addition, a number of major user organizations (e.g., the European Economic Community) have already embraced DCE as their standard for distributed applications.

30) What is the relationship between DCE and CORBA?

There is not a lot of direct relationship. DCE and CORBA are tools to help you build distributed systems. Each has its advantages and disadvantages. Use of one will not hinder future use of the other.

DCE provides a lower-level programming model than does CORBA. DCE is not fully "Object-Oriented". DCE has far better inter-operability than (current) CORBA products. DCE is an optional interoperability mechanism in the CORBA 2.0 specification.

In order to understand the relationship between DCE and the Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG), it is necessary to consider the past, the present and the future.

Past

Historically, the object paradigm has been viewed as a break with procedural styles of the past. Objects, which encapsulate data and procedures behind an external interface, are often contrasted with other approaches where procedures and data are treated separately.

In this context, DCE is a descendant of the procedural school which emphasizes the decomposition of programs into procedures and achieves distribution by locating some of those procedures remotely. Thus there was a tendency for the object community, including the OMG, to view DCE as technology which was obsolete before it was available.

However this view ignored the fact that designers of distributed systems had for a long time recognized that the most successful approach to developing distributed systems was to create encapsulated objects that can only be accessed via well defined interfaces. Thus the cornerstone of DCE RPC is the interface definition language (IDL) which allows the external attributes of a set of server operations to be specified.

Furthermore, the name-based binding mechanisms of DCE were extended to include the ability to bind to a server based on the object instances which it supports. These object binding mechanisms also allow the transparent selection among multiple implementations of the same server operations based on the type of the specified object. In object terminology this is called polymorphism.

The DCE notion of a server supporting interfaces consisting of one or more operations is so close to the notion of an object which provides one or more methods, that it should be no surprise that CORBA defies an IDL which differs from DCE IDL in only a few significant respects.

Principal among these is that in CORBA IDL every call must specify an object, which is used in determining the server to use. DCE can do this as well, but there is more work involved and it is optional. Another difference is that CORBA IDL allows an interface to be defined as an extension of one or more other interfaces, this is called interface inheritance. DCE does not permit interface inheritance, but may in the future. Implementation inheritance is not specified by either DCE or CORBA.

The use of object oriented techniques and principles should not be confused with using an object oriented language. Object oriented designs can be expressed in procedural languages, and in fact most of the current object environments supported C before supporting C++ or Smalltalk. Therefore, the fact that the DCE API is implemented in C is no barrier to using it to create a distributed object system. In fact, CORBA specified C language bindings first.

Present

CORBA should not be viewed in isolation, but in the context of all of the OMG's standardization efforts. OMG has defined a reference architecture (OMA) and has defined or is defining standards in a broad range of areas, including: databases, events, LifeCycle, transactions, persistence, security, naming and relationships. Viewed in this way, OMG's activities are much more ambitious and broader in scope than DCE.

A recent addition to CORBA, as a part of the CORBA 2.0 work was the definition of the means of interoperability between ORBs. CORBA 2.0 defines one mandatory and two optional mechanisms. The mandatory means is a new, lightweight protocol called UNO. The optional means are 1) via a gateway and 2) via an alternative protocol definition. At the present the only alternative protocol that has been defined is DCE RPC.

Many people who had hoped that DCE would be selected as the mandatory protocol were disappointed at this result. However, it should be observed that DCE is endorsed as alternative protocol and that several vendors have committed to providing ORBs that interoperate via DCE.

Another difference between DCE and the OMG standards is one of general philosophy. DCE has been defined quite rigorously in a series of documents published by X/Open. There is a set of conformance tests that are available to anyone. Any product passing these rigorous tests can be branded as DCE, without necessarily being based on the OSF code. Several vendors, including Microsoft and Tandem have reverse engineered significant portions of DCE.

OMG standards vary considerably in their level of detail, but in general, aim at a much looser level of standardization. In some cases, the standard merely specifies an object interface and some general semantics. This approach is a deliberate attempt to encourage diverse solutions which may be applicable in different environments. Even where specifications are relatively tight, for example in the area of CORBA portability, there is still room for considerable interpretation, as witness the fact that there is at least one company that provides consulting services on how to make CORBA applications compliant in practice.

At the present time, CORBA-compliant products and products that work with them do not provide a scaleable infrastructure suitable for large environments. Key features such as concurrency mechanisms, security and distributed transactions are not currently available. In contrast, DCE provides proven heterogeneous interoperability and most of the capabilities required by robust, production applications. Additional capabilities can be obtained by means of third products, such as transaction monitors built upon DCE. This situation will change over the next 2-3 years from a combination of standardization work by OMG and new product development by vendors.

Future

Most authorities agree that in the long term object technology will be the basis for building large scale distributed systems. In addition to the principle of encapsulation, object-based systems allow systems to be built up, evolve and be reconfigured as needed because of their ability to dynamically bind requesters to objects that provide services.

There are many specific issues concerning the properties of distributed object systems that are the subject of research and debate. It is also clear that there are some features of existing local object environments and languages that will not scale effectively to large scale distributed systems -- dynamic inheritance is one. Never the less, the general direction of the future is clear.

Clearly, the high level of interest in OMG defined standards comes not from current products, but from their exciting future potential. There is a natural tendency to compare DCE's current capabilities with the promise of CORBA's future. However, DCE is also evolving and will likely add additional object oriented features in the future. For example, HP is offering a DCE C++ class library which is expected to eventually become a standard part of DCE.

Where DCE was built by integrating existing software, OMG has chosen by and large to start with a clean sheet of paper. The idea is to be better able to implement object oriented constructs without the baggage of features carried over from previous systems. However, OMG faces great challenges. Object theory is currently in a great state of flux. Experts disagree on very fundamental issues about what features are necessary, useful or harmful.

Developing standards under these conditions is extremely challenging. OMG's approach to date has been to compromise and allow multiple alternatives. It is unclear whether this will succeed in the long run.

Does the conclusion that future distributed systems will be object-based mean that it is a mistake to build distributed systems today using DCE? The answer is no for several reasons. First, many organizations cannot afford to do nothing for several years. End users have pressing needs for

robust, scalable systems today. For many organizations, waiting would mean attempting to catch up with competitors who will have a tremendous head start.

Second, as this brief discussion has shown, it is possible to employ object techniques when developing distributed applications using DCE. Carefully designed systems will be able to take advantage DCE features such as dynamic binding and polymorphism and converge with CORBA-compliant systems as they mature.

Third, if object environments are to be successful in supporting industrial-strength distributed systems, they will have to address the problems that DCE addresses. The skills and techniques developed in working with DCE will be directly applicable to distributed systems environments of the future. This applies not only software developers, but also to operations personnel, planners, even business managers.

Further, the likelihood that DCE will be at least one technology for CORBA interoperability, implies that the eventually migration of applications which use DCE directly to an object environment should not present any insurmountable difficulties.

Finally, your direct experience in developing and operating robust distributed systems will provide you with great insight into the important characteristics of distributed systems environments as they apply to your organization's applications. This knowledge is vital to the shaping of successful tools of the future. History has shown that vendors and standards bodies, left to their own devices, will often miss the mark.

4.4 Common Facility Services

4.4.1 Email

4.4.1.1 Overview

This service provides development interfaces to manage sending electronic mail messages. The API is limited in scope as it only addresses sending a message from within an application. This will send a standard SMTP email message, nothing more nothing less.

4.4.1.2 Context

CSS is providing an object to manage the composing and sending of email. The ability to attach files is also provided. The object interfaced encapsulates the interface to OTS mail (SMTP).

4.2.1.3 Directives and Guidelines

MT Directives:

Any given mail object may only be used in one thread at a time, however multiple threads can concurrently access multiple mail objects. If you attempt to have two threads concurrently use the same mail object, the results are not defined.

Requirements:

In order to send a message the following methods must be called:

AddTo()

Subject()

And lastly Send()

(There is no requirement for a message body, although often one is quite useful)

Guidelines:

When adding headers, one should be careful not to specify ones which may conflict with default headers (Subject, From, To, Date, etc...)

When attaching files, the file is copied into memory so as to allow the file to be deleted once the call to AttachFile() has returned. As such, be aware of memory constraints imposed by attaching large files. (Binary files may be uuencoded upon reading so space consumed in memory may exceed a byte count of the file)

Be familiar with limits imposed by SMTP (such as line length and message length) and conform with those limits or, if violating them, be sure that the mail gateways sent through will be tolerant of your violations.

The destructor is the only method that will free up consumed resources and should be called once the message has been sent.

4.4.1.4 Sample Application Programmer Interface

Sample #1

Send a message.

Description

This code will instantiate a CsEmMailRelA object and use it to send both text and a file to root@eos.hitc.com.

```
CsEmMailRelA *mymessage = new CsEmMailRelA();//instantiate object
first
mymessage->AddTo("root@eos.hitc.com"); // set recipient
mymessage->AddMessage("\tThis message is from the sample code.\n"); // write some text
char *mybuffer = "While there is nothing wrong here,\nI just thought it would be nice to
drop a line\nand say hi.\n"; // make a character pointer
mymessage->AddMessage(mybuffer); // add it as well
mymessage->AddMessage("\nPlease look at the file attached below:\n"); // some more text
mymessage->AttachFile("/etc/passwd"); // include a file
// and a polite signature
```

```
mymessage->AddMessage("Sincerly,\n\nSample Code\n");
mymessage->Send(); // send the message
delete mymessage; // free up all those resources
```

4.4.1.5 Object Model

There is only the one CsEmMailRelA object involved. See the description of the object in the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.4.1.

4.4.2 FTP

4.4.2.1 Overview

FTP, which stands for File Transfer Protocol, may be used to send and receives files on a network. A program is run on the client machine which connects to a process on a server machine. The client may then send and receives files.

4.4.2.2 Context

CSS is providing an API to allow developers to instantiate a CsFtFTPRelA object and send the object messages controlling a ftp session. CSS will also provide kerberized FTP for systems which will support it. The CSS object is used to encapsulate FTP, an OTS standard application provided with most TCP/IP implementations.

4.4.2.3 Directives and Guidelines

MT Directives:

Any given ftp object may only be used in one thread at a time, however multiple threads can concurrently access multiple ftp objects. If you attempt to have two threads concurrently use the same ftp object, the results are not defined. Internally, it is required that the ftp object call fork() (or fork1() on the sparc) It will first lock the global mutex however, it is still possible that blocking system calls will be interrupted and return with EINTR and your code should be able to handle it, check the fork() man page for more details. However, to attempt to make things easier, you can instantiate an ftp object and connect to a machine at an arbitrary later time. (for example, you may wish to create the object before you spawn any threads)

Requirements:

Before you can transfer a file, you must call the following functions:

SetHostName()

SetUserName()

SetPassword()

Open()

You may then call any of the receiving or sending functions.

Guidelines:

Since each ftp object will have an attached pty, it is important not to consume too many of them since they are shared among *all* processes on the workstation.

You may close a connection, call any of the Set* functions and reopen a connection if you desire, all other parameters will be the same.

Files, passwords, and other strings may not have special characters like !|>" or others as part of their names as many of these characters have special meaning.

Unfortunately, due to the nature of ftp, it is impossible to determine what behavior should be taken when there is difficulty. (For example, if a network connection times out, should the application log an error and continue, or should it try to reconnect and continue where it left off?) As such the ftp object will only be able to throw errors when they are detected, but it is up to the application to determine what to do from there. A function CheckIfError() has been provided that will search the last message for any of the ftp error codes and will return the int value of the error found to aid the developer in processing the return text from ftp.

4.4.2.4 Sample Application Programmer Interface

Sample #1

File transfer

Description

This code will connect to a remote host, log in, change local directories, print out a listing of remote files. Then it will retrieve README (renaming it locally) and all files matching the regexp *.gz. Then it will log out. All of this will use a single FTP session.

```
CsFtFTPrelA my_ftp_connection;           // instantiate object
my_ftp_connection.SetHostName("unix22.andrew.cmu.edu");
my_ftp_connection.SetUserName("ftp");
my_ftp_connection.SetPassword("vschoema@eos.hitc.com");
my_ftp_connection.Open();                 // logs in
my_ftp_connection.SetLocalDirectory("destdir"); //lcd
cout << my_ftp_connection.GetListing();    // print listing
// Get a file, renaming it locally
my_ftp_connection.Receive("README", "remote-readme");
// Get all .gz files.
my_ftp_connection.ReceiveMatching("*.gz");
```

```
my_ftp_connection.Close();
```

4.4.2.5 Object Model

For more details on the object please see the CDR documentation, Release A CSMS Communications Subsystem Design Specification for the ECS Project, section 4.4.2.

4.4.3 Bulletin Board

4.4.3.1 Overview

This service provides development API for applications to send bulletin board messages. The API is limited in scope as it only addresses sending a message from within an application.

4.4.3.2 Context

CSS is providing an object to manage the posting of messages to NNTP Bulletin Boards. It is also possible to attach files a message before sending. The object encapsulates the interface to the COTS product.

4.4.3.3 Directives and Guidelines

MT Directives:

Any given BB object may only be used in one thread at a time, however multiple threads can concurrently access multiple BB objects. If you attempt to have two threads concurrently use the same BB object, the results are not defined.

Requirements:

In order to send a message the following methods must be called:

AddTo()

SetNNTPHost()

Subject()

And lastly Send()

(There is no requirement for a message body, although often one is quite useful)

Guidelines:

When adding headers, one should be careful not to specify ones which may conflict with default headers (Subject, From, To, Date, etc...)

When attaching files, the file is copied into memory so as to allow the file to be deleted once the call to AttachFile() has returned. As such, be aware of memory constraints imposed by attaching large files. (Binary files may be uuencoded upon reading so space consumed in memory may exceed a byte count of the file)

Be familiar with limits imposed by NNTP (such as line length and message length) and conform with those limits or, if violating them, be sure that the host sent to will be tolerant of your violations.

The destructor is the only method that will free up consumed resources and should be called once the message has been sent.

4.4.3.4 Sample Application Programmer Interface

This code will instantiate a CsBBMailRelA object and use it to send both text and a file to a newsgroup.

```
// instantiate object first.
CsBBMailRelA *mymessage = new CsBBMailRelA();
mymessage->AddTo("comp.soft-sys.dce");           // set recipient
// set nntp host
mymessage->SetNNTPHost("newsroom.hitc.com");
mymessage->AddMessage("\tThis message is from the sample code.\n"); // write some text
char *mybuffer = "While there is nothing wrong here,\nI just thought it would be nice to
drop a line\nand say hi.\n"; // make a character pointer
mymessage->AddMessage(mybuffer); // add it as well
mymessage->AddMessage("\nPlease look at the file attached below:\n"); // some more text
mymessage->AttachFile("/etc/passwd");             // include a file
// and a polite signature
mymessage->AddMessage("Sincerly,\n\nSample Code\n");
mymessage->Send();                               // send the message
delete mymessage;                               // free up all those resources
```

4.4.4 Virtual Terminal

There is no API or development work done through Virtual Terminal.

4.4.5 Event Logging

4.4.5.1 Overview

This service allows applications to log event and history information to a file which can later be used for fault, performance or statistical analysis. The service supports application defined events and collects management and fault data.

4.4.5.2 Context

CSS is providing several objects to allow applications to log events to an application specific file. In addition there are four management objects for logging management events (Fault, Performance, Audit, and Security) and these have the ability to send a SNMP trap.

4.4.5.3 Directives and Guidelines

MT Directives:

This object is fully threadsafe. Multiple threads may access the same event object concurrently, or multiple objects may access multiple event objects concurrently.

Directives:

To log a message you must begin with << EcStartMessage and must end with << EcEndMessage. They may span multiple lines (or even functions) of execution. However, once one thread starts logging a message to an object, no other thread will be able to log to that object until the corresponding EcEndMessage is sent.

The same thread that sent an object the EcStartMessage must also send the EcEndMessage.

While the application can retrieve the name of the log file, the application, under no circumstances, should in any way modify it.

Guidelines:

The only time one object will block another object is if they are both trying to write to the same log file at the same time. In that case, the first thread with the lockf() will write first, the other thread will wait until the first thread is done before writing. Internal deadlock should never be possible.

It will automatically log the filename and line number from the source code with each message.

4.4.5.4 Sample Application Programmer Interface

Sample #1

Log message

Description

This body of code will log to the error log associated with the application "testapp".

```
EcUtLoggerRelA mylog("testapp");  
mylog.SetErrorLevel(4) << EcStartMessage << "Here is a test message" << EcEndMessage;
```

4.4.5.5 Object Model

EcUtLoggerRelA object for application logging.

EcUtLoggerRelADebug object for logging application debug messages.

EcUtLoggerRelAMgmt object (virtual) for different management events. The objects described below inherit from the EcUtLoggerRelAMgmt class:

- EcUtLoggerRelAFault object for logging Fault events.
- EcUtLoggerRelAPerf object for logging Performance events.
- EcUtLoggerRelAAudit object for logging Audit events.
- EcUtLoggerRelASec object for logging Security events.

For additional information see the CDR documentation, Release A CSMS Communication Subsystem Design Specification for the ECS Project, section 4.4.5.

5. Distributed Object Framework (OODCE)

5.1 Service Description

The Distributed Object Framework (DOF) provides the mechanism for writing distributed applications in which one process may call the member functions of an object whose actual implementation resides in the address space of another process. To do this, the caller, or client, instantiates a "surrogate" object and provides it with some information that enables it to bind to the "manager" object running in a process somewhere on the network. The surrogate object presents a public interface that matches the public interface of the manager object. When a client calls a member function of the surrogate object, a remote procedure call (RPC) is actually made by the underlying framework to the corresponding member function of the manager object. To the client, it appears as though the call was executed locally.

DOF is provided by the Object-Oriented Distributed Computing Environment (OODCE). OODCE is a commercial product from Hewlett Packard which is built around the DCE development model. It is designed to enhance this model by extending the concept of interface definitions to C++ objects and to ease the learning curve by enabling developers to use "OODCE chips" for security, name service access, threads, etc. instead of having to be concerned with the lower level details of DCE.

Because OODCE is a commercial product, there is a set of documentation from the vendor that describes it in detail. In particular, the *HP OODCE C++ Library Programmer's Guide* should be read by anyone planning to use OODCE. This guide provides information about many of the features of OODCE through the use of examples and gives a basic application development summary in Appendix B. The *HP OODCE C++ Library External Reference Specification* provides a description of the purpose and public member functions of each OODCE class. The *OSF DCE Application Development Guide*, the *OSF DCE Application Development Reference*, and the *Guide to Writing DCE Applications* from O'Reilly provide detailed information about DCE, including the Interface Definition Language (IDL), which is also of interest to programmers using OODCE.

This Developer's Guide will not attempt to duplicate the detailed information already available in these documents. Rather, this guide will use an example to illustrate the main steps involved in creating a distributed application using DOF while providing some guidelines on how to use DOF effectively.

5.2 Why Use Distributed Objects?

In a distributed computing environment, it is common for separate processes executing on different computers to need to work together to accomplish a task. For example, a user may wish to locate a set of data based on some search criteria. In order to specify the search criteria and display the results of the search, the user may run a GUI-based client program on his or her own workstation. The request may initially be sent to a server program located at some central site.

To complete the search, this server program may itself need to send requests to other server programs located at other sites.

Distributed objects provide a convenient way for this communication to take place in an object-oriented environment. To the original user, it may appear as though the request has been satisfied by a single program running locally. In reality, however, the request has really been satisfied by several processes running anywhere on the network.

5.3 Example

Problem Definition:

In this example, a data server process, assumed to be running continuously somewhere on the network, provides access to a Data Server object. A client can request this Data Server object to create a View object. The client can then ask the View to conduct a search according to some criteria. As a result of conducting the search, the View will create a set of Esdt objects, each of which provides a set of commands that it can run on the data to which it is associated. If the client desires, it can ask the View at any time to prune the set of Esdt objects, deleting those that match a specific criteria. Then the client can iterate over the set of Esdt objects still held by the View and ask each Esdt object for the list of commands it can perform. After obtaining the list from an Esdt object, the client can ask that Esdt object to execute any of its commands. The client can ask the View to destroy the Esdt objects when it is done with them and can ask the Data Server object to destroy the View when it is done with it.

This example is instructive because it involves the basics of using OODCE to do remote procedure calls plus some more advanced techniques such as dynamic object creation and deletion. It also illustrates the use of the DCE Name Service, Security Service, and Access Control List (ACL) management from within OODCE. In addition, the example includes the use of threads on the client side to achieve a form of asynchronous communication with the server. This document will discuss the use of the Name Service and threads. While the example includes the code for using the Security Service and ACL management, these calls are described in detail in a companion Security Developer's Guide and so will not be covered in this Guide.

As we go through the example, we will provide some code fragments to illustrate points. You can find all of the code for the example (including security) in the Appendix at the end of this guide.

IDL Definition

The first step is to create the IDL files specifying the public interfaces for each of the distributed objects. In this example, there are three types of distributed objects: the Data Server object, a View object, and an Esdt object. These objects are distributed because their implementations will be running in the process space of the server, yet we will access their member functions from the client. You should create an IDL file for each distributed object specifying the member functions that you wish to be accessible from a client.

The easiest way to create an IDL file is to use the `uuidgen` command to generate a skeleton for the file with an interface UUID. If you run the command

```
uuidgen -i > DataServer.idl
```

you will create an idl file that already has an interface UUID defined and is ready for you to edit to add the public functions. The final IDL file for the DataServer object is shown below. The UUID becomes the unique identifier for the DataServer interface.

```
[
  uuid(bab991ea-7c04-11ce-b0cf-080009701906)
  version(1.0)
]
interface DataServer
{
  import "ObjRef.idl";

  DCEObjRefT *createView(
    [in] handle_t h
    );

  void destroyView(
    [in] handle_t h,
    [in] uuid_t  objUuid
    );
}
```

While you should refer to the OODCE/DCE documentation for a complete description of the Interface Definition Language, there are a few points that are worth emphasizing. First of all, the functions defined in the IDL file become the public member functions of the DataServer manager and client objects, minus the first handle_t argument of each function. You must supply this argument when you create the IDL file because the idl++ compiler requires that each remote procedure call use explicit binding management. However, OODCE uses this parameter "behind the scenes" so it does not appear in the idl++ generated object definitions.

Note the version number specified in the IDL file. This number (1.0) consists of a major version part, 1, and a minor version part, 0. The version number becomes a part of the interface information that is stored by the RPC runtime. When the runtime checks to make sure that a certain server can service a request made by a client, it requires that the major version number of the interface requested by the client equals the major version number of the interface offered by the server, and that the minor version number of the client interface is less than or equal to the minor version number of the server interface. This allows you to add functionality to a server

without requiring everyone using that server to get a new version of the client. For example, if you added an additional member function to the DataServer object above, you could keep the major version number at 1 and change the minor version to 1. Now users running old clients cannot access the new server functionality, of course, but they can still access all the old functionality without getting new copies of the client. However, if you changed the interface of an existing member function, for example by adding an additional parameter to createView, then you must change the major version number to 2. Now users running old clients will no longer be able to access the server. They must get new copies of the client.

Another important point to note is that you CANNOT use C++ objects as parameters of the member functions. OODCE provides enhancements to DCE to extend its notion of remote procedure calls to member functions of objects, but it has NOT extended the IDL language. You can only use the same data types as you could with DCE, which includes the base IDL data types, pointers, pipes, and C-style structures. (Consult the reference material for more details on the IDL language.) If you wanted to "pass" an object as a parameter, you would need to define a structure containing the object's member data, pass that structure, and then instantiate the object on the other side using the data in the structure. There are currently proposals submitted to the Open Software Foundation for C++ extensions to IDL to allow for objects to be passed as parameters of distributed member functions, but it is not known whether these proposals will become part of the OODCE/DCE standard.

It is worthwhile to look at the return parameter of the createView member function, which is a pointer to a DCEObjRefT. This structure is defined in the ObjRef.idl file which should be found in the /usr/include/oodce directory. It is "included" in this file by using the import directive. As you will see by looking at the idl file, a DCEObjRefT is not a pointer or a reference to an object in the C++ sense. Rather, it is just a structure containing an object UUID and some other structures which together provide the information necessary to bind to the referenced object. Therefore, passing an object reference in OODCE really means you are passing the information a client needs to locate and bind to the manager object represented by the reference.

The IDL files for the View and Esdt objects can be found at the end of this guide.

Compiling the IDL file

The next step is to run the idl files you have created through the idl++ compiler. Execute the command

```
idl++ DataServer.idl -keep source -I/usr/include/oodce
```

to compile the idl file shown above. The -keep source option tells the idl++ compiler to just generate and save the source code and not automatically invoke the C or C++ compiler. The -I option tells the idl++ compiler additional directories in which to search for any imported IDL files, in this case ObjRef.idl.

When you run this command, seven files will be generated: DataServerC.H and DataServerC.C are the definition and implementation files for the client-side surrogate object, while DataServer_cstub.c is a C stub that must be compiled with the C compiler and linked into a client application. DataServerS.H is the definition file for the server-side manager object; DataServerE.C is the entry point vector code that must be compiled with the C++ compiler and

linked into a server application, while `DataServer_sstub.c` is a C stub that must be compiled with the C compiler and linked into a server application. `DataServer.h` is an include file that is referenced by both the client and server C stubs and object definitions. This guide will discuss the surrogate and manager objects in some detail; for more information concerning the other files, refer to the reference material.

In this example, you would also run the `idl++` command on the `View` and `Esdt` IDL files.

Building the Server Application

A server application will typically consist of one or more manager objects that can be accessed by clients plus any additional objects created by the manager objects to help them do their jobs. While some manager objects may be created and destroyed dynamically while the application is running, there must always be at least one manager object that is created before the application begins listening for remote procedure calls. (If there were not such an object, a server would be useless, since no clients could communicate with it.)

In this example, the `DataServer` object is the manager object that is created when the application is brought up and it is the object to which clients initially bind and make their requests. The `DataServer` manager object in turn serves as an "object factory" to create `View` manager objects on behalf of clients. A client may then bind to a `View` manager object and direct further requests to it. The `View` manager objects serve as object factories to create `Esdt` objects as a result of executing some search criteria. A client can then bind to a specific `Esdt` object and make requests of it.

You may wonder why we would want to create individual `View` manager objects for each client request. Could not one `View` manager object handle requests from all the clients? While you can certainly design your server application this way, you may find it much simpler to create objects like the `View` manager that are only used by a single client, especially if these objects are going to hold state information that may depend on a particular client's requests. If you use a single manager object to service requests from all clients, then you must keep track of state for each client. Furthermore, each execution of a distributed member function is processed in a separate thread. If you have separate objects dedicated to each client, making the code thread-safe may be much easier.

The code fragment below shows the essential parts of the server main. (The `DataServerSrv` class is a derivation of the `DataServer` manager object generated by the `idl++` compiler, as explained below).

```
#include <oodce/Server.H>
#include <oodce/Pthread.H>
#include "DataServerSrv.h"
int main(int argc, char **argv)
{
    ...
}
```

```

DCEPthread *clean = new DCEPthread( theServer->ServerCleanup,
                                     NULL);

theServer->SetName((unsigned char*)cdsName);
theServer->SetGroupName((unsigned char*)groupName);
theServer->SetProfileName((unsigned char*)profileName);

DataServerSrv *obj = new DataServerSrv;

theServer->RegisterObject(*obj);

theServer->Listen();
}

```

The Global Server Object (GSO), `theServer`, is an instance of the `DCEServer` class provided by OODCE to manage objects and interact with DCE. There is only one instance of this class per process and you automatically get access to it by including the OODCE header file `Server.H`. You use member functions of the GSO to interact with DCE on behalf of the entire process, such as setting the process's directory service name, group, and/or profile, or registering the authentication service supported by the process with the DCE runtime. You also go through the GSO to register the necessary information with all the DCE subsystems for any manager objects that you create, as we will see below. CSS may provide a derived version of the GSO known as the Ecs Server Object (ESO) that will add additional functionality to the OODCE-provided `DCEServer` class. Some of this functionality will be transparent to the user, such as doing additional work when an object is registered with the ESO.

The first statement in the above code fragment starts a thread that sets up a signal handling function that waits for signals and does the appropriate cleanup (namely executes the GSO's `ServerCleanup` member function) upon process termination. This cleanup function will automatically remove entries from the name service and do other things to allow for a graceful termination. This is an example of the features that OODCE provides to free you from some of the details of working with DCE.

The `SetName` behavior of the GSO causes it to register its bindings in the Cell Directory Service (CDS), or name service, under the given name at the appropriate time (when `Listen` is invoked). Once the GSO is registered in the name service, clients can use its name to obtain its bindings and so do not have to know where this particular server is running. Calling the `SetName` behavior also causes the GSO to place certain information in the name space whenever a manager object is registered with it, as detailed below. If you call `SetName`, then you can also call `SetGroupName` and/or `SetProfileName` either before or after `SetName` is called, as in the

above example. Calling `SetGroupName` causes the server to be included as a member of the specified CDS group. A client surrogate object can then find this server by specifying only the group name. (This will cause the client to find any server in the group who exports the same interface as the surrogate object.) Similarly, calling `SetProfileName` causes the server to be included in the specified CDS profile, which is usually set to be `./:/cell-profile`. A client surrogate object can then find this server without specifying any name space information at all, because a search will automatically be made for the requested interface beginning at `./:/cell-profile`.

The next two statements create an instance of the `DataServer` manager object and register this object with the GSO. The OODCE base class constructors for manager objects automatically obtain and assign an object UUID for this instance. Registering an object tells the GSO to put it on some internal lists and enables the GSO to automatically register the object with the DCE runtime and name service at the appropriate time (when the `Listen` is invoked). This registration process causes a new entry to be made in the endpoint map on the machine where the server process is running. This entry consists of the manager object's interface UUID and object UUID, along with the binding information containing the protocol sequence, network address, and endpoint (port) of the server process. If the server process is using the name service, then the GSO will also add the manager object's interface UUID and, optionally, its object UUID to the server entry in the name space.

(Note: The object UUID will be added if the `RegisterObject` member function is called with its second parameter set to true. This parameter defaults to false. Normally, it is not necessary to export object UUIDs to the name space. However, because of the way information is stored in the name space and used to go to the endpoint map on a particular host and locate objects, you will want to export the object UUID for any object that clients will locate through the name space if 1) you are also using ACL management in your server or 2) you anticipate running several versions of your server, each registered under a different name in the name space, on the same machine.)

Every manager object created in a server application must be registered with the GSO for it to be accessible to clients. If it is registered before the GSO listen is done, as is the case in the server main, then it becomes accessible when the listen is executed. If it is registered after the listen, such as when a new manager object is dynamically created in response to a request to an existing manager object, then it becomes accessible as soon as it is registered.

Finally, the `Listen` member function of the GSO is called. The GSO will first register any information required by previous calls such as `RegisterObject()` or `SetName()` with the runtime, endpoint map, and/or the name service and then begin listening for remote procedure calls from clients for any objects that are currently registered with the GSO.

If you want more detailed information about any of the above calls, consult the reference material, in particular the *OODCE Programmer's Guide*.

The main guts of the server application consists of the implementations of the member functions defined in the IDL files for all the manager objects in the application. In this example, there are three manager objects - the DataServer, the View, and the Esdt. The code fragment below shows a part of the DataServerS.H file generated by the idl++ compiler:

```
class DataServer_1_0_ABS : public virtual DCEObj, public DCEInterfaceMgr {
public:
    ...
    // some constructors
    ...

    virtual DCEObjRefT *createView(
    ) = 0;

    virtual void destroyVoid(
        /*[in]*/ uuid_t objUuid
    ) = 0;
};

class DataServer_1_0_Mgr : public DataServer_1_0_ABS {
public:
    ....
    // some constructors
    ...

    virtual DCEObjRefT *createView(
    );

    virtual void destroyView(
        /* [in] */ uuid_t objUuid
    );
};
```

Note that there are two classes defined here: a `DataServer_1_0_ABS` class and a `DataServer_1_0_Mgr` class. In the ABS class, the member functions defined in the IDL file are declared as pure virtual, while in the Mgr class they are not. The `DataServer_1_0_Mgr` class is meant to be the manager object. The idea here is that the `idl++` compiler has provided the declarations of the `createView` and `destroyView` member functions and the developer needs only to provide a source file implementing `DataServer_1_0_Mgr::createView` and `DataServer_1_0_Mgr::destroyView`.

However, most manager objects are also going to contain some protected or private data members and/or member functions that serve as helpers to implement the public member functions. These must also be included in the class definition. One option is to edit the `idl++` generated file above and add these data or function members to the `DataServer_1_0_Mgr` definition. In practice, though, it turns out that you will probably regenerate the `idl` files several times as you decide to add more public behavior or to modify the parameters of the public functions. Each time you do this, you will need to add the additional data or functions again, which can get tedious, especially if there are quite a few of them.

Another solution, which was used in this example, is to derive your own manager class directly from the `DataServer_1_0_ABS` class and provide the actual implementations of the public member functions as well as any protected member functions or data in this derived class. Then you declare instances of this derived class to be your manager objects rather than instances of `DataServer_1_0_Mgr`. Now if you add or change public member functions in the IDL file, you don't need to edit the generated file. You simply make the corresponding changes in your manager class.

The `DataServerSrv` class serves as the data server manager class in this example. A portion of its definition is show below.

```
class DataServerSrv : public DataServer_1_0_ABS {
public:
    .... some constructors...

    virtual DCEObjRefT *createView();

    virtual void destroyView(uuid_t objUuid);

protected:
    ... member data and helper member functions...
};
```

This class contains the implementations of the public member functions `createView` and `destroyView` as well as the declarations and/or implementations of any protected/private data or member functions. Its class definition is in file `DataServerSrv.h` and its implementation is in file `DataServerSrv.cc`. The same technique has been used for the `View` and the `Esdt` manager objects, creating the derived classes `ViewSrv` and `EsdtSrv` from the `idl++` generated ABS classes.

It is worthwhile to examine in more detail some aspects of the implementation of `createView` and `destroyView`. The following code fragment shows the essential parts of `createView`:

```
DCEObjRefT *DataServerSrv :: createView()
{
    ViewSrv *newView = new ViewSrv;
    ....
    theServer->RegisterObject(*newView);

    return(newView->GetObjectReference());
}
```

The first statement creates a new instance of the `ViewSrv` manager object. The OODCE base class constructors automatically obtain and assign an object UUID to this instance. The next statement registers this new object with the GSO, which puts it on its internal lists and does whatever is necessary to make it accessible to clients. Since the GSO is already in the listen state, this includes immediately adding the `View` interface UUID to the runtime and the server entry in the name service (if it is not already there) and making a new entry in the endpoint map on the server host machine for this `View` manager object. Finally, the `GetObjectReference` member function of a manager object is invoked and the result is returned as a `DCEObjRefT` pointer. This member function actually returns a reference to the class `DCEObjectReference`, which in turn has a conversion operator to a `DCEObjRefT` pointer. How the information contained in the `DCEObjRefT` is used by the client will be described in the next section.

The next code fragment shows the implementation of the `destroyView` member function.

```
void DataServerSrv :: destroyView(uuid_t objUuid)
{
    theServer->UnregisterObject(&objUuid);

    theServer->RemoveObjects();
}
```

This member function is invoked by a client with the object UUID of a View manager object. The first statement causes the GSO to remove the object with the given UUID from its internal lists of registered objects and also to remove the object's entry in the runtime, the endpoint map or the name service, if required. According to the OODCE documentation, the object is then placed on a GSO internal list of objects that can be deleted. The RemoveObjects member function causes the GSO to call the destructors of all objects on this list. One important point should be made. You should probably always call the GSO RemoveObjects member function to delete objects you have unregistered rather than delete them yourself. Since these objects remain on an internal list after they are unregistered, you may be leaving dangling pointers in the GSO if you explicitly delete them. The OODCE documentation implies that the GSO may call RemoveObjects from time to time automatically to do garbage collection, so it could be fatal to delete an unregistered manager object yourself instead of going through the GSO.

The full implementations of the DataServer, View, and Esdt manager objects are included with the code at the end of this guide. These implementations, along with the main and the OODCE generated server side code, are compiled and linked together to make the server application.

To run the server in this example, you must supply the name under which to register it in the name service, along with the principal name and keytab file necessary for it to establish its own login context and register its authentication service. (See the security Developer's Guide for more information on security.) Therefore, to run the Sun version of the server with a name service name of `./:/ims/ds1` and a principal name of `fred` where the secret key is in the keytab file `fred.pwd`, you should enter

```
srv -cdsName ./:/ims/dataServer -princName fred -keyFile fred.pwd
```

from the command line. The HP version of the server is started by invoking `srv_hp` with the same parameters.

Building the Client Application

Normally, building a client application involves instantiating the OODCE generated surrogate objects as needed, with some information to enable them to bind to their manager counterparts running somewhere on the network, and making calls to their member functions. These calls to the surrogates' member functions are turned by the underlying framework into remote procedure calls which get dispatched to the manager objects to which the surrogates are bound. The OODCE client side surrogates come ready to use as is from the `idl++` compiler.

However, there may be times when it is desirable to "wrap" the client surrogates to make them easier to use by the client developers. A case can be made that such wrapping ought to be done by the server developers, since they are the ones who present the client developers with the interfaces to the server objects. This example presents a case where wrapping can be useful.

The DataServer manager object can be asked to create a View for a client, who then wishes to make requests directly to that View manager object. The return parameter from the `createView` member function is a pointer to a `DCEObjRefT` structure which contains the information the

client needs to bind to the View manager object that was created. Upon receiving this information, the client needs to instantiate a client side View surrogate object and bind it to the manager object. Then the client can call the member functions of the View surrogate and have them executed by the manager View object. This sequence needs to be done each time the client makes a createView call.

It would be nice from the client's perspective if this sequence were encapsulated into the DataServer surrogate object. Then a call to the DataServer surrogate createView member function would return a pointer to a View surrogate object already bound to its corresponding manager object. The client only needs to invoke the member functions of the View surrogate. All of the "ugly" details have been hidden from the client. Furthermore, if the DataServer were just another object in the same address space as the client, its createView member function would almost certainly return a pointer to a View object.

It is possible to make it appear to the client that the createView member function really returns a pointer to a View object by deriving a DataServer surrogate class that has this behavior from the idl++ generated surrogate. The code fragment below shows the createView and destroyView member functions of DataServer_1_0, the generated surrogate object whose definition and implementation is in files DataServerC.H and DataServerC.C

```
class DataServer_1_0 : public DCEInterface {
public:
    .... constructors...

    DCEObjRefT *createView(
    );

    void destroyView(
        /* [in] */ uuid_t objUuid
    );
};
```

The next code fragment shows part of the definition of the DataServer_cl class, which is derived from DataServer_1_0.

```
class DataServer_cl : public DataServer_1_0 {
public:
    ... constructors ...
```

```

        virtual View_cl *createView();

        virtual void destroyView( View_cl*);
};

```

This derived class surrogate contains `createView` and `destroyView` member functions that return and take pointers to a `View_cl` object respectively, instead of returning a `DCEObjRefT` pointer and taking an object UUID, as in the base class surrogate. (The `View_cl` class is derived from `View_1_0` just like `DataServer_cl` is derived from `DataServer_1_0`.) Notice that these member functions have the same names as the member functions in the base class surrogate, and so they hide the base class member functions. (Since `createView` is not declared virtual in the base class, it is not an error for the derived class `createView` to have a different return type than the base class `createView`.)

The essential parts of `DataServer_cl :: createView` are shown in the following code fragment.

```

View_cl *DataServer_cl :: createView()
{
    DCEObjRefT *ref = DataServer_1_0::createView();

    DCEUuid uuid(&(ref->objId));
    View_cl *view = new View_cl( DCEGetBindingHandle(ref),
                                uuid);

    free(ref);

    return view;
};

```

The first statement makes the call to the base class `createView` member function, which is actually a remote procedure call to the `DataServer` manager object running in the server. This object creates a new `View` manager object in the server and returns a pointer to a `DCEObjRefT`, which, you recall, contains the information a client needs to bind to the new `View` manager object.

The next two statements show how this binding is done. An instance of a `DCEUuid` is created and assigned the object UUID of the `View` manager object, which was passed as part of the

DCEObjRefT structure. Then a View surrogate object is instantiated and given, in its constructor, the binding handle information from the DCEObjRefT structure plus the object UUID of the manager object. The binding information tells the surrogate the network address of the machine where the server is running. It can then use the endpoint map on that machine and the object UUID to determine the endpoint (port) of the server and bind to the manager object. Since the View surrogate knows the interface UUID for the View interface, it can check that the object it is binding to really does support that interface. Of course, all of this is done transparently by OODCE/DCE. Notice that it was not necessary to go to the name service to complete this binding.

Next, the memory allocated to hold the DCEObjRefT structure is freed. This memory was allocated automatically by the DCE client stub to hold the structure that was returned by the remote procedure call. It needs to be freed by the caller of the RPC when it is done with the data to avoid memory leaks. See the reference material for more information on dealing with pointers as parameters and return values of RPCs. Finally, the pointer to the newly created View_cl surrogate is returned to the caller.

In summary, by encapsulating the above sequence in a derived surrogate class, we are freeing the client from having to repeat this same sequence everywhere it calls the createView member function. We are also presenting a more natural C++ interface that corresponds to the interface that the client would expect if the DataServer manager were really a local object. In addition, we are hopefully avoiding some potential problems such as forgetting to free memory allocated by DCE by doing it once in the derived surrogate code.

The next code fragment shows the implementation of the DataServer_cl :: destroyView member function, which takes a pointer to a View_cl surrogate.

```
void DataServer_cl :: destroyView( View_cl *view)
{
    if( view)
    {
        view->doneWithEsdts();

        uuid_t uuid = view->srvObjUuid();
        DataServer_1_0::destroyView(uuid);

        delete view;
    }
}
```

The first statement invokes a member function defined in the View interface, telling the View manager object to which this surrogate is bound to destroy any Esdt objects it may have created. Next, the object UUID of this manager object is obtained and then the destroyView member function defined in the DataServer interface is invoked with this UUID. If you look at the definition and implementation of the View_cl class at the back of this guide, you will notice that srvObjUuid() was added as a member function of the derived class to access some protected data of the OODCE generated View_1_0 class. Finally, the View surrogate object is destroyed.

Another interesting member function is View_cl::getNextEsdt(), which is shown in the following code fragment.

```
Esdt_cl *View_cl :: getNextEsdt()
{
    Esdt_cl *rtnPtr = 0;
    DCEObjRefT *ref = View_1_0::getNextEsdt();
    if( ref)
    {
        DCEUuid uuid(&(ref->objId));
        if( myCurEsdt)
        {
            myCurEsdt->SetBinding( DCEGetBindingHandle( ref));
            myCurEsdt->SetServerObject( uuid);
        }
        else
        {
            myCurEsdt = new Esdt_cl( DCEGetBindingHandle( ref),
                                     uuid);
        }
        free(ref);
        if( myCurEsdt)
        {
            rtnPtr = myCurEsdt;
        }
    }
}
```

```

        return rtnPtr;
    }

```

After the return pointer is initialized to 0, the getNextEsdt member function of the View base class surrogate is called. This is an RPC which causes the remote View manager object to advance through its list of Esdt managers and return a "reference" to the next object in its list. The View_cl class keeps a pointer called myCurEsdt to an Esdt_cl surrogate. If an instance of this surrogate has already been instantiated, the View_cl class simply changes the binding information and object UUID of the surrogate to match the information returned in the reference structure. This will cause the Esdt_cl surrogate to rebind itself to the Esdt manager object represented by the reference. If the Esdt_cl surrogate has not yet been instantiated, then the View_cl class creates one, giving it the binding information and object UUID in its constructor. Finally, the reference storage is freed and the return pointer is set to the current Esdt_cl instance. Of course, this technique of keeping just one Esdt_cl instance on the client side is purely arbitrary. You could choose to create as many Esdt_cl surrogates as there are manager objects on the server, and bind each of them to a different manager object.

Once any derived client surrogates have been implemented, the rest of the client application consists of using these surrogates (or using the OODCE generated surrogates directly) to access the services of the manager objects running in the server as they are needed. This example includes a Motif-based application that demonstrates the use of the surrogate objects. There is also a non Motif-based "simple" client that just consists of a main which uses the surrogates. Since the Motif code tends to hide the use of the surrogates, the following code fragment demonstrating a client application is taken from the simple client.

```

#include "DataServer_cl.h"
#include "View_cl.h"
#include "Esdt_cl.h"

int main(int, char **argv)
{
    ....

    DataServer_cl ds((unsigned char*)srvCdsName);

    View_cl *view = ds.createView();
    if( view)
    {

```

```

        view->search();
        view->prune();

        Esdt_cl *curEsdt;
        while( curEsdt = view->getNextEsdt())
        {
            ... call esdt member functions...
        }
        view->doneWithEsdts();
        ds.destroyView( view);
    }
    return 0;
}

```

The first statement instantiates the DataServer surrogate with the CDS name of the server as an argument. When a surrogate is instantiated in this way, it automatically goes out to the name service to get the binding information that has been registered for that name. Since the DataServer surrogate knows the DataServer interface UUID, it will check that the server is actually providing an object that implements that interface and then it will bind to that object. Recall from the previous section that a client always binds first to the DataServer manager object.

The client then invokes the createView member function, which returns a pointer to a View_cl surrogate bound to a newly created View manager object in the server. Using this surrogate, the client then asks the View to search and prune, creating and modifying a list of Esdt manager objects in the server. Finally, the client uses the getNextEsdt member function to iterate through the list of Esdt objects, each time receiving an Esdt_cl surrogate bound to the next Esdt manager object in the list. The client can then make RPC calls to that object through the surrogate. Finally, the client tells the View to destroy its list of Esdt objects, and then tells the DataServer to destroy the View. At this point, only the DataServer manager object is left in the address space of the server (assuming no other client has created View or Esdt objects in the server, of course).

The important thing to note is that the simple client does not need to worry about binding handles, DCEObjRefT structures, or object UUIDs. Since all these details have been encapsulated into the derived surrogate classes, the client code using these surrogates becomes much simpler and more natural, as though the manager objects were local to the client.

The full implementations of the DataServer, View, and Esdt derived surrogate classes are included with the code at the end of this guide. These implementations, along with the OODCE

generated client side code, are compiled and linked with either the non Motif-based main or the Motif-based main and Motif application class to make the client applications.

Both the Motif-based client and the simple client require the CDS name of the server and the principal name under which the server is running (for security). To run the Sun version of the clients, where you want them to communicate with a server registered in the CDS with the name `./:ims/dataServer` and running under principal name fred, enter either

```
mclient -srvCdsName ./:ims/dataServer -srvPrincName fred
```

for the Motif client or

```
sclient -srvCdsName ./:ims/dataServer -srvPrincName fred
```

for the simple client from the command line. The HP versions of the clients are called `mclient_hp` and `sclient_hp`.

Using Threads To Achieve Asynchronous Communication

The Motif-based version of the client provides an example of using threads to achieve a form of asynchronous communication with the server. After making a request to an Esdt manager object for its list of possible commands, you are presented with a dialog allowing you to select a command and ask the Esdt manager object to execute that command. The dialog also gives you the option of selecting all of the commands to be executed. If you choose this option, the Motif client uses threads to send all of the requests to the Esdt manager object simultaneously. This allows the client to dispatch all of the requests to the server without waiting for any of them to finish. Though each individual RPC to the server is synchronous, the fact that the call is made in a separate thread makes it asynchronous to the client process as a whole.

To facilitate the use of threads for asynchronous communication, CSS is providing a function called `EvalThread` and an abstract class called `AsyncCom`, which have both been used in this example. This function and class are designed to work with the OODCE-provided `DCEPthread` class which you use to start and control the execution of a thread.

An instance of a `DCEPthread` represents a thread of execution but it is not the thread itself. It provides member functions supporting things such as joining with the thread, changing the thread's priority or scheduling policy, or setting its initial stack size. OODCE provides other threads classes for using mutex locks, setting attributes for use in multiple threads, using thread-specific storage, and using condition variables. For more detailed information on using threads in OODCE, you should consult the reference material.

Basically, there are two ways to start an independent thread using the `DCEPthread` class. You can instantiate an instance of this class, passing a `DCEPthreadProc` and a `DCEPthreadParam` as parameters to the constructor, as shown below.

```
DCEPthread aThread( proc, /*a DCEPthreadProc*/  
                    param /* a DCEPthreadParam */)
```

This will immediately launch a new thread executing the function `proc`, passing it the parameter `param`. That is, a `DCEPthreadProc` must be either a static member function of a class or a function that is not a member of any class which has the signature

```
DCEPthreadResult proc( DCEPthreadParam);
```

Both `DCEPthreadResult` and `DCEPthreadParam` are typedef'd as `void*`, allowing you to pass any object or data structure containing the information `proc` needs during its execution. Of course, the one who launches the thread executing `proc` must know the kind of data `proc` is expecting.

You can also instantiate an instance of the `DCEPthread` class with no parameters, set some attributes if desired, and then launch the execution of the thread by calling the `Start` member function with a `DCEPthreadProc` and a `DCEPthreadParam`, as shown below.

```
DCEPthread aThread;  
... set some attributes...  
aThread.Start( proc, /* a DCEPthreadProc */  
              param /* aDCEPthreadParam */);
```

The CSS-provided `EvalThread` function is a non-member function which you pass as the `DCEPthreadProc` parameter when starting the thread. You must then pass a concrete instance of a class derived from `AsyncCom` as the `DCEPthreadParam` because this is the parameter that `EvalThread` will be expecting. The following code fragments show the definition of the `AsyncCom` abstract class and the implementation of the `EvalThread` function.

```
class AsyncCom {  
public:  
    AsyncCom();  
    virtual ~AsyncCom();  
  
    virtual void SetData( void *data);
```

```

virtual void *GetData();

virtual int PreInvoke() = 0;
virtual int Invoke() = 0;
virtual int PostInvoke() = 0;

protected:
    void *myData;
};

DCEPThreadResult EvalThread( DCEPThreadParam p)
{
    AsyncCom *acom = (AsyncCom*)p;

    if( acom->PreInvoke())
    {
        acom->Invoke();
    }
    acom->PostInvoke();

    delete acom;
    return 0;
}

```

You derive a class from AsyncCom, overriding the PreInvoke, Invoke, and PostInvoke member functions to perform whatever operations you desire. You may also use the SetData and GetData member functions to set and retrieve a pointer to some data that you wish to use in the overridden Invoke member functions. You then instantiate and start a thread, giving it EvalThread as the procedure to execute and your instance of a derived AsyncCom class as the parameter. EvalThread will then begin executing in a separate thread, calling your overridden Invoke functions and then deleting the AsyncCom instance to clean up.

The Motif-based client provides a concrete example of the use of this class. We derive a class called CliAsyncCom from AsyncCom. An instance of a class called ExecCmdParams will be stored as the data member of this class. ExecCmdParams contains a pointer to the MotifCliAppl

object and the command string to be executed. Then, when the user selects "All" as the command option to be executed, we iterate through all the command strings, launching a separate thread to execute each of them simultaneously. The code fragment below shows how we start each thread.

```
void MotifCliAppl :: executeAllCmds()
{
    ... determine number of strings in dialog...

    for each string in dialog (except "All" string)
    {
        set cmd equal to the string

        ExecCmdParams *params = new ExecCmdParams;
        params->appl(this);
        params->command(cmd);

        CliAsyncCom *com = new CliAsyncCom;
        com->SetData((void*)params);

        DCEPthread aThread;

        aThread.Termination(Pthread_detach_on_delete);

        aThread.Start( EvalThread,
                      (DCEPthreadParam(com));
    }
}
```

Determining the number of strings and obtaining each string involve some X/Motif calls, which are shown in the code in the Appendix. For each string obtained, we create a new instance of the ExecCmdParams class, calling the appl member function to set a pointer back to this MotifCliAppl object and setting the command string to the current string. Then we create a new

instance of the CliAsyncCom class, setting its data to be the ExecCmdParams object. Finally, we start a new thread of execution with EvalThread as the function to be executed and the CliAsyncCom object as the parameter to be passed to it. Setting the termination attribute of aThread to Pthread_detach_on_delete means that aThread will be deleted when its scope is exited, while the execution thread itself will continue to execute.

The code fragment below shows the implementation of the CliAsyncCom destructor and Invoke member function.

```
CliAsyncCom :: ~CliAsyncCom()
{
    if( myData)
    {
        ExecCmdParams *params = (ExecCmdParams*)myData;
        delete params;
    }
}
int CliAsyncCom :: Invoke()
{
    if( myData)
    {
        ExecCmdParams *params = (ExecCmdParams*)myData;
        MotifCliAppl *appl = params->appl();
        appl->executeSpecificCmd( params->command());
    }
    return 1;
}
```

Note that the ExecCmdParams data is deleted in the CliAsyncCom destructor. Alternatively, this data could have been deleted in the PostInvoke member function, but, in any case, it is important to remember to clean this data up to avoid memory leaks.

In the Invoke member function, the myData parameter is first cast to an ExecCmdParams object. From this object, the pointer to the MotifCliAppl object is retrieved, and then its executeSpecificCmd member function is called with the command string. This is the same member function of MotifCliAppl that is called when the user selects a single command string from the dialog. It invokes the executeCmd member function of the current Esdt object held by the MotifCliAppl object. It also takes care of cleaning up the storage allocated for the command string when it was obtained from the dialog. For this reason, we chose to reuse it here instead of storing a pointer to the current Esdt surrogate object in ExecCmdParams and repeating the call to its executeCmd member function and the necessary cleanup.

6. System Management

6.1 User Profile External Interfaces

The following classes are provided as external interfaces in support of the MSS User Profile services.

MsUsProfile

MsAcGuestUserReq

Each of these classes are defined within this document, along with usage and example code.

CLASS: MsUsProfile

DESCRIPTION: The MsUsProfile class embodies all attributes of a single user profile, along with the locating, retrieval, and storage of the user profile.

USAGE:

MsUsProfile provides the application programmer set and get functions for each object attribute, and the following functions for modifying the user profile database:

UpdateUserProfile ()

This function will update the database containing the user profile information. The only requirement is the user profile must already exist.

Populate ()

Based on the current userid, this method will retrieve the values of the user profile from the user profile database, and fill the respective user profile attributes.

EXAMPLE CODE:

```
// example code modifies a user profile
void main ( void )
{
    // instantiate a user profile object
    MsUsProfile UserProfile;

    // retrieve the user profile
    UserProfile.SetUserID ("jdoe_ecscell");
    UserProfile.Populate ( );
}
```

```

// change the user profile telephone number
UserProfile.SetTelNumber ("301-925-1000");

// commit the changes to the user profile database
UserProfile.UpdateUserProfile ();
}

```

CLASS: MsAcGuestUserReq

DESCRIPTION: The MsAcGuestUserReq class embodies all the attributes and functionality of a user request for an ECS account.

USAGE:

MsAcGuestUserReq provides the application programmer set and get functions for each object attribute, and the following functions for modifying the user profile database:

AddUserRequest ()

Adds the user request to the user request database.

EXAMPLE CODE:

```

// this example adds a new request to the user request database
void main ( void )
{
// instantiate a guest user request object
MsAcGuestUserReq UserRequest;

// set the user request values
UserRequest.SetName ( "John Doe" );
UserRequest.SetTelNumber ( "301-925-1000" );

// add the user request to the user request database
UserRequest.AddUserRequest ( );
}

```

6.2 Management Agent Services External Interfaces

The management services provided by MSS are made available to ECS subsystems through a set of public classes as mentioned section 1.1 of this document (Management Service Description).

These public classes provide the interface for ECS subsystems to MSS and vice versa. These public classes, their attributes and their methods are covered in this section. They are however, described in more detail in the Interface Control Document (DID 313).

The following classes are provided as external interfaces to the management subagent for ECS application programmers.

EcAgManager
MsAgMonitor
EcAgEvent

Each of these classes are defined within this document, along with usage and example code.

CLASS: EcAgManager

DESCRIPTION: The EcAgManager class provides mapping facilities between the ECS application and the management agent. Callback registration functions are provided by the class for the retrieval of application specific metrics (variable values). These need to be used by programmers in order to register the callbacks for this functionality. Likewise, application programmers may register a callback function for the shutdown event.

USAGE:

EcAgManager provides the application programmer one function for use in their application. They are as follows:

RegisterCallBack (int ActionType, MssCallBackFuncPtr *FuncPtr);

This function is used by the application programmer to register callback functions. The first arguments is the action in which the callback should be executed. The second function is a pointer to the actual callback function

The following are legal action types:

GET_PERFORMANCE
GET_FAULT
GET_CONFIG
SHUTDOWN

The callback function must be of the following type

typedef int (*MssCallBackFuncPtr)(void *pValue, int nIndex);

The application programmer should take note that their callback function may be called at any time after the registration of the callback and before the shutdown callback is issued. Since EcAgManager is running in a separate thread of execution, the programmer must also take any steps within the callback function to protect any critical code sections (via semaphores, file locks, ...).

Different data structures are passed in the pValue attribute depending on the action type. These structures are the following:

```
GET_PERFORMANCE      typedef struct {
                        char  szPerfType [17];
                        int   nPerfValue;
                        int   nPerfThreshold
                      } CBD_PERFORMANCE;

GET_FAULT             typedef struct {
                        char  szFaultType [17];
                        int   nFaultValue;
                      } CBD_FAULT;

GET_CONFIG            typedef struct {
                        char  szCfgParam [17];
                        char  szCfgValue [17];
                      } CBD_CONFIG;

SHUTDOWN              int nNow;
```

EXAMPLE CODE: This example provides sample callbacks (shutdown, performance) and their registration with the EcAgManager class.

```
int MyShutdownCallback ( void *pValue; int nIndex )
{
    int nNow = (int)*pValue;

    if ( nNow ) {
        printf ("Shutting down immediatly\n");
        exit (0);
    }
}
```

```

}
else {
    printf ("Shutting down later\n");
    bShutdownLater = TRUE;
}

return 0;
}

int MyPerformanceGetCallback ( void *pValue; int nIndex )
{
    CBD_PERFORMANCE* pPerfData = (CBD_PERFORMANCE*) pValue;
    int nReturnCode = 0;

    // critical code section, so use semaphore locking
    SEMAPHORE::Lock ( MGMTLOCK);
    switch ( nIndex ) {
        case NUMBER_WIDGETS_PROCESSED_INDEX:
            pValue->nValue = nNumberOfWidgetsProcessed;
            break;
        case NUMBER_WIDGETS_CACHED:
            pValue->nValue = nNumberOfWidgetsCached;
            break;
        default:
            nReturnCode = 1; // no more performance attributes, so return 1
    }
    SEMAPHORE::UnLock ( MGMTLOCK );

    return nReturnCode;
}

```

```

int main ( void )
{
    EcAgManager MsManager;

    MsManager.RegisterCallback ( GET_PERFORMANCE, MyPerformanceGetCallback );
    MsManager.RegisterCallback ( SHUTDOWN, MyShutdownCallback );

    // normal application code goes here

    return 0;
}

```

CLASS: MsAgMonitor

DESCRIPTION: The MsAgMonitor class provides application programmers the ability to start and stop the management agents' monitoring of a specified process (transient processes included). Both DCE and non-DCE based application may be monitored in this fashion. Between the StartMonitor request and the StopMonitor request, in the case of a fault, a notification is sent to the specified recipient.

USAGE:

MsAgMonitor provides the application programmer two functions for use in their application. They are as follows:

```
StartMonitor ( int nPID, char *szDestination, char *arg3 );
```

```
StartMonitor ( char *arg1, char *szUUID, char *arg3 );
```

The function start monitor has been overloaded to account for both DCE and non-DCE applications. The first instance of the method will instruct the management agent to monitor an application with the provided process ID. The second instance of the method instructs the management agent to monitor a DCE application with the provided UUID.

EXAMPLE CODE:

```
int MyClass::MySpawningOfAProcess ( )
```

```

{
  int nPID;

  // start a child application to produce widgets
  nPID = SpawnWidgetCreator ( );

  // instruct the management agent to monitor the child process
  MsMonitor.StartMonitor ( nPID, NULL, NULL );
}

```

CLASS: EcAgEvent

DESCRIPTION: The EcAgEvent class provides application programmers the ability to send application events to the management subagent.

USAGE:

EcAgEvent provides the application programmer one functions for use in their application. It is as follows:

LogEvent (char *szParentID, char *szMyId, int nEventType, int nSeverity, int nDisposition, char *szMessage)

The application programmer calls this function each time a reportable event has occurred. nEventType specifies the event type. nSeverity specifies the severity of the even. nDisposition specifies the disposition of the event. szMessage provides a brief description of the event.

EXAMPLE CODE:

```

MyClass::MyErrorHasOccured ( )
{
  // an error has occured, so report it to the management agent
  myId = EcEvent.GetMyId;
  EcEvent.LogEvent ( parentId, myId, FAULT, WARNING, NULL,
                    "cannot find widget initialization file" );
}

```

This page intentionally left blank.

Abbreviations and Acronyms

ACL	Access Control List
AFS	Andrews File System
AI&T	Algorithm Integration and Test
AIT	Algorithm Integration Team
ANSI	American National Standards Institute
API	Application program (or programming) interface
ASCII	American Standard Code for Information Exchange
ATM	Asynchronous Transfer Mode
ARP	Address Resolution Protocol
ASTER	Advanced Spaceborne Thermal Emission and Reflection Radiometer
BB	Bulletin Board
BBS	Bulletin Board Service
BIND	Berkeley Internet Name Domain
BGP	Border Gateway Protocol
BOA	Basic Object Adapter
CAC	Command and Activity Controller
CCB	Change Control Board (Hughes Convention)
CCB	Configuration Control Board (NASA Convention)
CCR	Configuration Change Request
CDS	Cell Directory Service
CDR	Critical Design Review
CDRL	Contract data requirements list
CERES	Clouds and Earth's Radiant Energy System
CIDR	Classless Interdomain Routing
CM	Configuration management
CMAS	Configuration Management Application Service
CMIP	Common Management Information Protocol
CNE	Campus Network Environment

CORBA	Common object request broker architecture
COTS	Commercial off-the-shelf (hardware or software)
CPU	Central processing unit
CSMS	Communications and System Management Subsystem
CSS	Communication Subsystem
DAAC	Distributed Active Archive Center
DADS	Data Archive and Distribution System
DB	Database
DBMS	Database management system
DCE	Distributed computing environment (OSF)
DEC	Digital Equipment Corporation
DECOM	FOS Decommutation Process
DFS	Distributed File System
DID	Data item description
DME	Distributed Management Environment
DNS	Directory Name Service
DOF	Distributed Object Framework
DPR	December Progress Review
DS	Data Server (FOS)
DTS	Distributed Time Server (part of DCE)
ECS	EOSDIS Core System
EDOS	EOS Data and Operations Center
EDF	ECS Development Facility
E-Mail	Electronic Mail
EMC	Enterprise Monitoring and Coordination
EOC	EOS Operations Center (ECS)
EOS	Earth Observing System
EOSDIS	Earth Observing System Data and Information System
EP	Evaluation Prototype
ESN	EOSDIS Science Network
EPV	Endpoint Vector

FDDI	Fiber distributed data interface
FDF	Flight Dynamics Facility
FOS	Flight Operations Segment
Ftp	File Transfer Protocol
GB	Gigabyte (10^9)
GCDIS	GDS Global Directory Service
GDS	Global Directory Service
GSFC	Goddard Space Flight Center
GUI	Graphic user interface
HAIS	Hughes Applied Information Systems (ECS)
HiPPI	High Performance Parallel Interface
HP	Hewlett Packard
Http	Hyper Text Transfer Protocol
I/F	Interface
I&T	Integration & Test
IBM	International Business Machines, Inc.
ICD	Interface control document
ICMP	Internet Control Messaging Protocol
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IR-1	Interim Release 1
ISO	International Standards Organization
ISO+	IsoCELL (Isolation Cell)
ISS	Internetworking Subsystem of CSMS
IST	Instrument Support Toolkit
IST	Instrument Support Terminal
Kerberos	Security protocol developed by MIT; base for DCE security
Kftp	Kerberized file transfer protocol
KLOC	Kilolines (10^3) of code

Ktelnet	Kerberized telnet
LAN	Local area network
LaRC	Langley Research Center
LLC	Logical Link Control
LOC	Lines of code
LSM	Local System Management
M&O	Maintenance and operations
MBONE	Multicast Backbone
MIB	Management Information Base
MIME	Multimedia Internet Mail
MLM	Mid-Level Manager
MOPITT	Measurement of pollution in the troposphere
MOSPF	Multicast Open Shortest Path First
MR-AFS	Multi-Resident Andrew File System
MSFC	Marshall Space Flight Center
MSS	Systems Management Subsystem
MUI	Management User Interface
NCR	Non-conformance Report
NFS	Network file system
NIC	Network Interface Card
NNTP	Network News Transfer Protocol
NOAA	National Oceanic and Atmospheric Administration
NOLAN	Nascom Operational Local Area Network
NSI	NASA Science Internet
NTP	Network Time Protocol
OA	Off-Line Analysis Process
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
OMG	Object Management Group
OMT	Object Modelling Technique
OO	Object-oriented

OODCE	Object-oriented DCE
OODBMS	Object-oriented database management system
ORB	Object Request Broker
OS	Object Services (CSS building blocks)
OSF	Open Software Foundation
OSI	Open System Interconnect
OSI-RM	OSI Reference Model
OSPF	Open Shortest Path First
PAC	Privilege Attribute Certificate
PDR	Preliminary Design Review PDR-A
PDU	Protocol Data Unit
PPP	Point-to-Point Protocol
POSIX	Portable Operating System Interface for Computer Environments
PSC	Pittsburgh Supercomputing Center
PTGT	Privilege Ticket Granting Ticket
RDBMS	Relational database management system
RFA	Remote File Access
RFC	Request for comments
RIP	Routing Information Protocol
RMA	Reliability, Maintainability, Availability
RMON	Remote Monitoring
RMP	Reliable Multicast Protocol
RPC	Remote procedure call
RTS	Real-Time Server (FOS)
SCF	Science Computing Facility
SDPF	Sensor Data Processing Facility
SDR	Software/System Design Review
SDR	Sensor data record
SGI	Silicon graphics
SLOC	Source lines of code
SMC	System Monitoring and Coordination

SMDS	Switched Multi-megabit Data Service
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network
SQL	Simple Query Language
TBD	To be determined
TCP/IP	Transmission Control Protocol/Internet Protocol
TGT	Ticket Granting Ticket
TMN	Telecommunications Management Network
TRMM	Tropical Rainfall Measurement Mission
TSDIS	TRMM Science Data Information System
UDP	User Datagram Protocol
UIOAR	User Interface Off-Line Analysis Request Window
URL	Universal Resource Locator
US	User Station (FOS)
UUID	Universal Unique Identifier
UTC	Universal time code
V0	Version 0
VT	Virtual Terminal
WAN	Wide area network
WWW	World Wide Web
X	X Protocol
X.500	OSI standard for directory services (207)
XDS	X/Open Directory Service
XFN	X/Open Federated Naming
XOM	X/Open OSI-Abstract-Data Manipulation