4. Communications Subsystem (CSS) Design

4.1 Introduction

This section gives an introduction of all the CSS services whose detailed design description is 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, an Object Model, Dynamic Models (Scenarios), and implementation details (functional model as required). Table 4.1-1 provides a road map for the CSS Design 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
Generic Security Service API (GSSAPI)	4.2.8
Distributed Object Framework	4.3
Common Facility Services	4.4
Electronic Mail	4.4.1
FTP	4.4.2
DFS	4.4.3
Bulletin Board	4.4.4
Virtual Terminal	4.4.5
Event Logger	4.4.6
Bulk Data Service	4.4.7
INFRASTRUCTURE Services	4.5
Process Framework Service (PF)	4.5.1
Server Request Framework (SRF)	4.5.2
Subscription Service	4.5.3
Universal Reference	4.5.4

The CSS in Release B is being developed on the formal track for its DCE related components, while some new components are developed on incremental track.

Distributed Computing Environment (DCE) from the Open Software Foundation (OSF) is a CSS baseline COTS product for releases Ir1, A and B. 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. Currently missing in this package is the accommodation of C++ language class library bindings. During December Progress Review (DPR) and Segment Design Review (SDR) preparation, ECS adopted a long-range architecture based on Common Object Request Broker Architecture (CORBA) 2.0 ORB services, object services, and common-facility services. At SDR, many of these services were not yet available, so ECS adopted a phased approach: a communications infrastructure based on DCE in Release A and a migration to CORBA 2.0 in Release B. At the Preliminary Design Review (PDR), it was apparent that CORBA 2.0 releases were still in the future and thus the migration to CORBA was pushed out beyond Release B. Release A's communications subsystem (CSS) would encapsulate DCE so as to simplify the change from DCE to CORBA. This strategy simultaneously recognized:

- DCE is today's state-of-the-art solution for industrial-strength, vendor-independent, distributed enterprise computing. It is a widely available, multi-vendor client/server communications environment integrated with security and directory services.
- CORBA plus affiliated object and common-facility services, if successful in the market place, can offer significant advantages over DCE to the long-term GDS Global Directory Service (GCDIS)/UserDIS architecture. It will be an environment with multi-vendor interoperable Object Request Brokers (ORBs) with object passing and dynamic bindings.

A key CSS CDR-A goal was thus to decide how to accomplish its DCE-to-CORBA migration. At SDR CSS conservatively proposed a full custom solution; between SDR and PDR, CSS undertook extensive prototyping and product studies to drive an engineering trade study to select an optimal DCE encapsulation approach.

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.

Recently OSF has announced plans to incorporate into DCE an object passing method (XIDL from DEC and the OODCE from HP) in its release 1.2 of DCE. This announcement may provide an alternative migration path for ECS. In addition HP, DEC and IBM have announced plans to implement CORBA 2.0 over DCE. These events reinforce the original decision to use DCE for ECS at Release A, and help to reduce the migration costs for ECS later.

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.

The CSS Implementation Summary, Table 4.1-2, takes into account the use of DCE encapsulation (using custom APIs, OODCE and DCE). Custom code estimates for CSS to provide the required services for Release Ir1 and A reflect the use of OODCE and reflect a lower combined line of code (LOC) estimate than what was presented at the CSMS SDR.

Table 4.1-2. CSS Implementation (1 of 3)

Services	**Ir1 or Release A or Release B**	**COTS1 (WGlue) Custom Code COTS2 (COTS upgrade)**	**CI**	**Notes**
OBJECT SERVICES				
Directory/ Naming	Ir1	COTS	DCCI	BIND -- DNS; CDS -- DCE
	Release A	Custom Code	DCCI	GDS -- X.500
	Release B	COTS2	DCCI	Upgrade DCE to 1.1 version
Security	Ir1	COTS	DCCI	Authentication (reuse of APIs from EP4).
	Release A	COTS + Custom Code	DCCI	Login Service. Remaining Features (include Authorization and Access Control)
	Release B	COTS2+Custom Code	DCCI	Upgrade DCE to 1.1 version
Message Passing	Ir1	COTS	DCCI	Synchronous
	Release A	COTS + Custom Code	DCCI	Asynchronous & Deferred Synchronous.
	Release B	COTS2	DCCI	Upgrade DCE to 1.1 version
Thread	Ir1	COTS	DCCI	COTS within DCE & OODCE
	Release B	COTS2	DCCI	Upgrade DCE to 1.1 version
Time	Ir1	COTS + Custom Code	DCCI	DCE UTS; internally developed code required as enhancements & glue code

Table 4.1-2. CSS Implementation (2 of 3)

Services	Ir1 or Release A or Release B	COTS1 (WGlue) Custom Code COTS2 (COTS upgrade)	CI	Notes
	Release B	COTS2+Custom Code	DCCI	develop glue code for FOS external time box using the Time Provider Interface within DCE
LifeCycle (Initialization /Activation)	Release A	COTS + Custom Code	DCCI	OODCE COTS + Custom Code required as enhancement and glue code.
	Release B	COTS2+Custom Code	DCCI	develop code to communicate with mode management/Process Framework
DISTRIBUTED OBJECT FRAMEWORK (DOF)				
DOF Services	Ir1	COTS	DCCI	OODCE
	Release A	COTS	DCCI	OODCE
	Release B	COTS2	DCCI	OODCE
COMMON FACILITY SERVICES				
Electronic Mail	Ir1	COTS + Custom Code	DCCI	Mail packages bundled with the operating system, ELM or Pine Is Not Elm (PINE) which are available in public domain. Custom API for application support.
	Release A	Custom Code	DCCI	Sending messages to a Title.
	Release B	COTS2	DCCI	
File Access	Ir1	COTS + Custom Code	DCCI	File Transfer -- ftp, kftp. Custom File Transfer API.
	Release A	COTS + Custom Code	DCCI	Remote File Access. Custom code for scheduling file transfers.
	Release B	COTS2+Custom Code	DCCI	
DFS	Release B	COTS	DCCI	Transparent remote file access.
Bulletin Board (BB)	Ir1	COTS	DCCI	Reuse BB from EPs.
	Release A	COTS	DCCI	Based on NNTP & use public domain packages (rn, xrn).

Table 4.1-2. CSS Implementation (3 of 3)

Services	Ir1 or Release A or Release B	COTS1 (WGlue) Custom Code COTS2 (COTS upgrade)	CI	Notes
	Release B	COTS2	DCCI	
Virtual Terminal	Ir1	COTS	DCCI	Telnet, kTelnet, X (for viewing graphics), Dial-Up service.
	Release B	COTS2	DCCI	Upgraded DCE and OS
Event Logger	Ir1	Custom Code	DCCI	Reuse of APIs from EP4
	Release A	Custom Code	DCCI	Class Library
	Release B	COTS2	DCCI	
INFRASTRUCTURE SERVICES				
Process Framework	Release B	COTS2+Custom Code	DCCI	
Server Request Framework	Release B	COTS2+Custom Code	DCCI	
Subscription Service	Release B	Custom Code	DCCI	
Universal Reference	Release B	Custom Code	DCCI	

1 COTS (W/Glue)—COTS requires some "glue" code and APIs for encapsulating OODCE

2 COTS(Upgrade DCE) ---Upgrade DCE version from 1.0.3 to 1.1 with OODCE and Operating System upgrade also

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 would ordinarily need to use the XDS/XOM interface in order to store and retrieve application specific information into the namespaces but 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.

A new private class CsDcXds has been added which contains methods that allow for the storage and retrieval of information into and from the CDS database. It implements the DCE XDS/XOM application programming interfaces.

The MSS Management interface provide a way to startup and shutdown the DNS and the X.500 namespaces. The M&O staff will use this interface that invokes an API which resides in the Distributed Object Framework to manage the startup and shutdown functionality.

The CDS and GDS consists of entries (name and attribute value pairs). These entries may be protected through Access Control Lists (authorization). While lookup for the local principal is done through the CDS, the lookup for foreign principals is done through GDS/DNS.

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.

In Release B, DCE 1.1 will be used. As a result two new DCE CDS features will be available. Cells can be nested hierarchically and cell can have multiple names (known as aliasing). These two features should be helpful in cell management.

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. This information is stored in the Directory Services DBMS and is accessible through the X.500 and the DNS Namespaces. 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, or an Universal Reference (UR) 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. This interface will be designed and implemented by the CIDM Subsystem.

4.2.1.3 Object Model

See Figure 4.2.1.3-1.

4.2.1.3.1 CsDcXds Class

Parent Class:Not Applicable

Attributes:

None

Operations:

CsDcXds - Default Constructor.
Arguments:
Return Type:Void
Privilege:Public

CsXdsAddAttr - Add an attribute to the entry.
Arguments:a_fullEntryName[]:EcTchar,a_eltType:EcTInt,a_attrValue[]:EcTchar
Return Type:int
Privilege:Public

CsXdsAddEntry
Arguments:a_fullEntryName[]:EcTChar,a_eltType:EcTInt,a_attributeValue[]:EcTC har

CsXdsAddValue - Add a value to the attribute.
Arguments:a_fullEntryName[]:a_eltType:EcTInt,a_attrAddValue[]:EcTchar
Return Type:EcTInt
Privilege:Public

CsXdsDelEntry - Delete the address entry from the CDS. The call to this function will return a '1' if it succeeded, and '0' if it failed.
Arguments:a_fullEntryname[]:EcTChar
Return Type:int
Privilege:Public

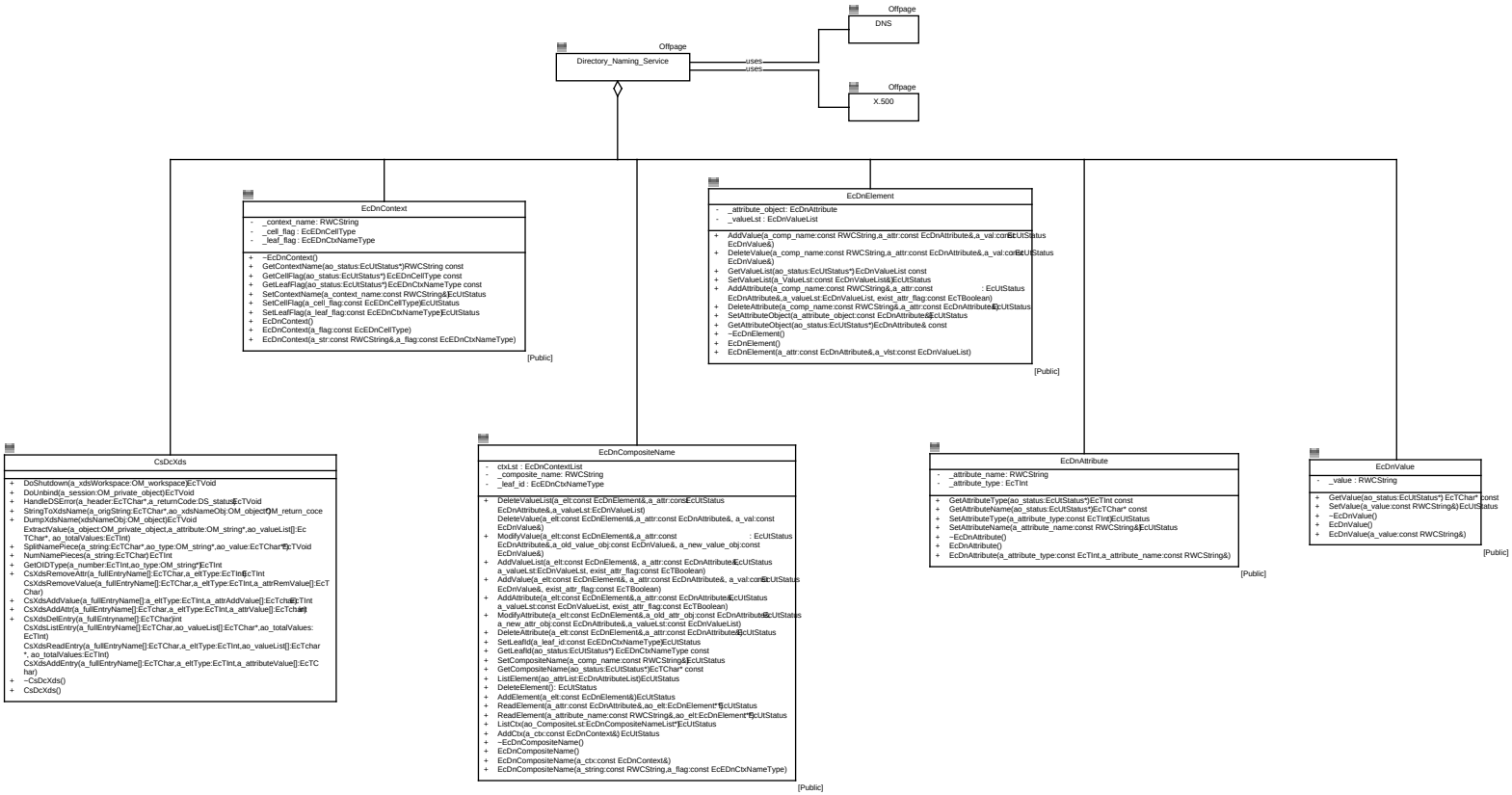

Figure 4.2.1.3-1. Naming Object Model

CsXdsListEntry
Arguments:a_fullEntryName[]:EcTChar,ao_valueList[]:EcTChar*,ao_totalValues: EcTInt

CsXdsReadEntry
Arguments:a_fullEntryName[]:EcTChar,a_eltType:EcTInt,ao_valueList[]:EcTchar *, ao_totalValues:EcTInt

CsXdsRemoveAttr - Remove an attribute from an entry.
Arguments:a_fullEntryName[]:EcTChar,a_eltType:EcTInt
Return Type:EcTInt
Privilege:Public

CsXdsRemoveValue
Arguments:a_fullEntryName[]:EcTChar,a_eltType:EcTInt,a_attrRemValue[]:EcT Char

DoShutdown - Free the workspace.
Arguments:a_xdsWorkspace:OM_workspace
Return Type:EcTVoid
Privilege:Public

DoUnbind - Unbind the session.
Arguments:a_session:OM_private_object
Return Type:EcTVoid
Privilege:Public

DumpXdsName - Dumps a full xds name
Arguments:xdsNameObj:OM_object
Return Type:EcTVoid
Privilege:Public

ExtractValue
Arguments:a_object:OM_private_object,a_attribute:OM_string*,ao_valueList[]:Ec TChar*, ao_totalValues:EcTInt

GetOIDType - Return the XDS/XOM OID type of te given attr.
Arguments:a_number:EcTInt,ao_type:OM_string*
Return Type:EcTInt
Privilege:Public

HandleDSError - Extracts the error number from a DS_status return code, prints it in an error message, then terminates the program
Arguments:a_header:EcTChar*,a_returnCode:DS_status
Return Type:EcTVoid
Privilege:Public

NumNamePieces - Returns the number of pieces in a string name.
Arguments:a_string:EcTChar
Return Type:EcTInt
Privilege:Public

SplitNamePiece - Divides a piece of a name (a_string) into its XDS attribute type and value.
Arguments:a_string:EcTChar*,ao_type:OM_string*,ao_value:EcTChar**
Return Type:EcTVoid
Privilege:Public

StringToXdsName - Convert a string that is a DCE name to an XDS name object (class DS_C_DS_DN)
Arguments:a_origString:EcTChar*,ao_xdsNameObj:OM_object*
Return Type:OM_return_coce
Privilege:Public

~CsDcXds - Default Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The CsDcXds class has associations with the following classes:
Directory_Naming_Service (Aggregation)

4.2.1.3.2 DNS Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Implementation of DNS

Attributes:

None

Operations:

None

Associations:

The DNS class has associations with the following classes:
Class: Directory_Naming_Service uses

4.2.1.3.3 Directory_Naming_Service Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
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.

Attributes:

None

Operations:

None

Associations:

The Directory_Naming_Service class has associations with the following classes:
Class: DNS uses
Class: X.500 uses

4.2.1.3.4 EcDnAttribute Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The EcDnAttribute class will contain an attribute name, and type. It will also be referenced by the EcDnElement class. This class will provide methods to get the attribute name and type.

Attributes:

_attribute_name - Attribute name
Data Type:RWCString
Privilege:Private

Default Value:

_attribute_type - Attribute type.
Data Type:EcTInt
Privilege:Private
Default Value:

Operations:

EcDnAttribute - Default Constructor
Arguments:
Return Type:Void
Privilege:Public

EcDnAttribute
Arguments:a_attribute_type:const EcTInt,a_attribute_name:const RWCString&
Return Type:Void
Privilege:Public

GetAttributeName - This operation gets the attribute name of the object.
Arguments:ao_status:EcUtStatus*
Return Type:EcTChar* const
Privilege:Public

GetAttributeType - This operation gets the attribute type of the object.
Arguments:ao_status:EcUtStatus*
Return Type:EcTInt const
Privilege:Public

SetAttributeName
Arguments:a_attribute_name:const RWCString&
Return Type:EcUtStatus
Privilege:Public

SetAttributeType
Arguments:a_attribute_type:const EcTInt
Return Type:EcUtStatus
Privilege:Public

~EcDnAttribute - Default destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcDnAttribute class has associations with the following classes:
Directory_Naming_Service (Aggregation)

4.2.1.3.5 EcDnCompositeName Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The EcDnCompositeName class will define 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 will provide 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 element.

Attributes:

_composite_name - Nested set of contexts.
Data Type:RWCString
Privilege:Private
Default Value:

_leaf_id - Leaf flag: 0 = A Context Name 1 = An Entry Name (leaf)
Data Type:EcEDnCtxNameType
Privilege:Private
Default Value:

ctxLst - Context object.
Data Type:EcDnContextList
Privilege:Private
Default Value:

Operations:

AddAttribute
Arguments:a_elt:const EcDnElement&,a_attr:const EcDnAttribute&, a_valueLst:const EcDnValueList, exist_attr_flag:const EcTBoolean
Return Type:EcUtStatus
Privilege:Public

AddCtx
Arguments:a_ctx:const EcDnContext&

Return Type:EcUtStatus
Privilege:Public

AddElement
Arguments:a_elt:const EcDnElement&
Return Type:EcUtStatus
Privilege:Public

AddValue - This operation adds a value to the value list.
Arguments:a_elt:const EcDnElement&, a_attr:const EcDnAttribute&, a_val:const EcDnValue&, exist_attr_flag:const EcTBoolean
Return Type:EcUtStatus
Privilege:Public

AddValueList
Arguments:a_elt:const EcDnElement&, a_attr:const EcDnAttribute&, a_valueLst:EcDnValueLst, exist_attr_flag:const EcTBoolean
Return Type:EcUtStatus
Privilege:Public

DeleteAttribute
Arguments:a_elt:const EcDnElement&,a_attr:const EcDnAttribute&
Return Type:EcUtStatus
Privilege:Public

DeleteElement - This method is used to delete an element (attribute, value list) from the Directory Service.
Arguments:
Return Type:EcUtStatus
Privilege:Public

DeleteValue
Arguments:a_elt:const EcDnElement&,a_attr:const EcDnAttribute&, a_val:const EcDnValue&

DeleteValueList
Arguments:a_elt:const EcDnElement&,a_attr:const EcDnAttribute&,a_valueLst:EcDnValueList
Return Type:EcUtStatus
Privilege:Public

EcDnCompositeName - Default Constructor.
Arguments:
Return Type:Void
Privilege:Public

EcDnCompositeName - Constructor number two. It will be used in the case in which DCE names are stored directly into the DCE Directory Service (DNS/GDS). It takes a string which is the full Directory Service cell name. This can be used in place of using the first constructor and adding on to that composite name via the add_ctx method.
Arguments:a_string:const RWCString,a_flag:const EcEDnCtxNameType
Return Type:Void
Privilege:Public

EcDnCompositeName
Arguments:a_ctx:const EcDnContext&
Return Type:Void
Privilege:Public

GetCompositeName - This method is used to retrieve the composite name of the object.
Arguments:ao_status:EcUtStatus*
Return Type:EcTChar* const
Privilege:Public

GetLeafId - This method is used to retrieve the leaf id.
Arguments:ao_status:EcUtStatus*
Return Type:EcEDnCtxNameType const
Privilege:Public

ListCtx - This method is used to list the contents of the composite name in the Directory Service (soft links, object entries). Initiates the enumeration process for the target context. It returns a handle to a ECSRWCtxListP object (RW) used to enumerate the immediate subordinates of the named entry.
Arguments:ao_CompositeLst:EcDnCompositeNameList*
Return Type:EcUtStatus
Privilege:Public

ListElement - This method is used to list the contents of the composite name (entry name) in the Directory Service. Using the composite name (distinguished name) and the attribute type, this method will return a list of attributes corresponding to the entry given.
Arguments:ao_attrList:EcDnAttributeList
Return Type:EcUtStatus
Privilege:Public

ModifyAttribute
Arguments:a_elt:const EcDnElement&,a_old_attr_obj:const EcDnAttribute& a_new_attr_obj:const EcDnAttribute&,a_valueLst:const EcDnValueList
Return Type:EcUtStatus
Privilege:Public

ModifyValue
Arguments:a_elt:const EcDnElement&,a_attr:const EcDnAttribute&,a_old_value_obj:const EcDnValue&, a_new_value_obj:const EcDnValue&
Return Type:EcUtStatus
Privilege:Public

ReadElement
Arguments:a_attr:const EcDnAttribute&,ao_elt:EcDnElement**
Return Type:EcUtStatus
Privilege:Public

ReadElement
Arguments:a_attribute_name:const RWCString&,ao_elt:EcDnElement**
Return Type:EcUtStatus
Privilege:Public

SetCompositeName
Arguments:a_comp_name:const RWCString&
Return Type:EcUtStatus
Privilege:Public

SetLeafId - This method is used to set the leaf id.
Arguments:a_leaf_id:const EcEDnCtxNameType
Return Type:EcUtStatus
Privilege:Public

~EcDnCompositeName - Default Destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcDnCompositeName class has associations with the following classes:
Directory_Naming_Service (Aggregation)

4.2.1.3.6 EcDnContext Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The EcDnContext class defines the path/set of bindings with distinct atomic names. Every context has an associated naming convention. An EcDnContext object is passed to the

EcDnCompositeName object in a structural form as an ordered sequence of components

Attributes:

_cell_flag - Flag:0 = global root name (/...), 1 = cell root name (/.:)
Data Type:EcEDnCellType
Privilege:Private
Default Value:

_context_name - Context name (a partial name of the distinguished name)
Data Type:RWCString
Privilege:Private
Default Value:

_leaf_flag - Flag:0 = directory path, 1 = entry path (leaf/object), 2 = Any other type.
Data Type:EcEDnCtxNameType
Privilege:Private
Default Value:

Operations:

EcDnContext - Default constructor.
Arguments:
Return Type:Void
Privilege:Public

EcDnContext - Constructor number one. This constructor will be used to define the type of name to be used in the DCE environment (global root name or cell root name type)
Arguments:a_flag:const EcEDnCellType
Return Type:Void
Privilege:Public

EcDnContext - Constructor number two. This constructor will be used to define a context that can be used to build the composite name.
Arguments:a_str:const RWCString&,a_flag:const EcEDnCtxNameType
Return Type:Void
Privilege:Public

GetCellFlag - This method will return the cell flag
Arguments:ao_status:EcUtStatus*
Return Type:EcEDnCellType const
Privilege:Public

GetContextName - This method will return the context name
Arguments:ao_status:EcUtStatus*

Return Type:RWCString const
Privilege:Public

GetLeafFlag - This method will return the leaf flag
Arguments:ao_status:EcUtStatus*
Return Type:EcEDnCtxNameType const
Privilege:Public

SetCellFlag - This method will set the cell flag
Arguments:a_cell_flag:const EcEDnCellType
Return Type:EcUtStatus
Privilege:Public

SetContextName
Arguments:a_context_name:const RWCString&
Return Type:EcUtStatus
Privilege:Public

SetLeafFlag - This method will set the leaf flag
Arguments:a_leaf_flag:const EcEDnCtxNameType
Return Type:EcUtStatus
Privilege:Public

~EcDnContext - Default Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcDnContext class has associations with the following classes:
Directory_Naming_Service (Aggregation)

4.2.1.3.7 EcDnElement Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The EcDnElement class will contain an element, which is an attribute-value list pair. It will be referenced by the EcDnCompositeName class. This class will provide methods to add value(s), get value list, delete value(s), modify value(s) and get the element name.

Attributes:

_attribute_object - Attribute object.
Data Type:EcDnAttribute
Privilege:Private
Default Value:

_valueLst - RW list of value objects
Data Type:EcDnValueList
Privilege:Private
Default Value:

Operations:

AddAttribute
Arguments:a_comp_name:const RWCString&,a_attr:const EcDnAttribute&,a_valueLst:EcDnValueList, exist_attr_flag:const EcTBoolean
Return Type:EcUtStatus
Privilege:Public

AddValue
Arguments:a_comp_name:const RWCString,a_attr:const EcDnAttribute&,a_val:const EcDnValue&
Return Type:EcUtStatus
Privilege:Public

DeleteAttribute
Arguments:a_comp_name:const RWCString&,a_attr:const EcDnAttribute&
Return Type:EcUtStatus
Privilege:Public

DeleteValue
Arguments:a_comp_name:const RWCString,a_attr:const EcDnAttribute&,a_val:const EcDnValue&
Return Type:EcUtStatus
Privilege:Public

EcDnElement - Default Constructor
Arguments:
Return Type:Void
Privilege:Public

EcDnElement
Arguments:a_attr:const EcDnAttribute&,a_vlst:const EcDnValueList
Return Type:Void
Privilege:Public

GetAttributeObject - This operation reads/gets the value list of the element set by the user which will be used when adding the element into the Directory Name space.
Arguments:ao_status:EcUtStatus*
Return Type:EcDnAttribute& const
Privilege:Public

GetValueList - This operation reads/gets the value list of the element set by the user which will be used by add_element() when an entry is added. This value list is in memory (a RW list type).
Arguments:ao_status:EcUtStatus*
Return Type:EcDnValueList const
Privilege:Public

SetAttributeObject
Arguments:a_attribute_object:const EcDnAttribute&
Return Type:EcUtStatus
Privilege:Public

SetValueList
Arguments:a_ValueLst:const EcDnValueList&
Return Type:EcUtStatus
Privilege:Public

~EcDnElement - Default Destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcDnElement class has associations with the following classes:
Directory_Naming_Service (Aggregation)

4.2.1.3.8 EcDnValue Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The EcDnValue class defines a value. It will also be referenced by the EcDnElement and EcDnAttribute class.

Attributes:

_value - Value name
Data Type:RWCString
Privilege:Private
Default Value:

Operations:

EcDnValue - Default Constructor
Arguments:
Return Type:Void
Privilege:Public

EcDnValue
Arguments:a_value:const RWCString&
Return Type:Void
Privilege:Public

GetValue - This method will return the value.
Arguments:ao_status:EcUtStatus*
Return Type:EcTChar* const
Privilege:Public

SetValue
Arguments:a_value:const RWCString&
Return Type:EcUtStatus
Privilege:Public

~EcDnValue - Default destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcDnValue class has associations with the following classes:
Directory_Naming_Service (Aggregation)

4.2.1.3.9 X.500 Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Implementation of X.500

Attributes:

None

Operations:

None

Associations:

The X.500 class has associations with the following classes:
Class: Directory_Naming_Service uses

4.2.1.4 Functional Model

The 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 else where. There will be a root context, with no parent, through which all the contexts can be accessed. Leaf contexts form as 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.

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.

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, the user must first 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.

Table 4.2.1.4-1 summarizes the Directory Naming Service classes which are discussed in detail in the following subsections. Figure 4.2.1.4-1 pictorially describes the CDS entry structure.

4.2.1.5 Dynamic Model Scenarios

4.2.1.5.1 Scenario #1

4.2.1.5.1.1 Abstract

This scenario describes how to list the contents of the Directory Service.

4.2.1.5.1.2 Interfaces

Cell Directory Service

4.2.1.5.1.3 Stimulus

The user enters a series of atomic names in order to create a composite name that is in accordance with the CDS naming conventions, and makes a request to list the contents of that directory.

4.2.1.5.1.4 Desired Response

The user will receive the contents of the composite name.

4.2.1.5.1.5 Participating Classes

EcDnContext, EcDnCompositeName.

Table 4.2.1.4-1. Naming Service Object Responsibility Matrix

Class Name	Description
EcDnContext	The context 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 ECSCompositeName 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 ECSCompositeName class will define 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 will provide 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 will contain an element, which is an attribute-value list pair. It will be referenced by the EcDnCompositeName class. This class will provide methods to add value(s), get value list, delete value(s), modify value(s) and get the element name.
EcDnAttribute	The attribute class will contain an attribute name, and type. It will also be referenced by the ECSElement class. This class will provide methods to get the attribute name and type.
EcDnValue	The EcDnValue class will contain a value. It will also be referenced by the EcDnElement and EcDnAttribute class.

4.2.1.5.1.6 Pre-conditions

The composite name must be established.

4.2.1.5.1.7 Post-conditions

This doesn't change the state of the naming database. It only retrieves information about the composite name.

4.2.1.5.1.8 Scenario Description

Create a context root name. Define the type of name to be used in the 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. Proceed to list the entries/ directories in that composite name, that is, using the composite name object call the method 'listCtx'.

4.2.1.5.1.9 Event Trace

4.2.1.5.2 Scenario #2

4.2.1.5.2.1 Abstract

This scenario describes how to add an element into the Directory Service.

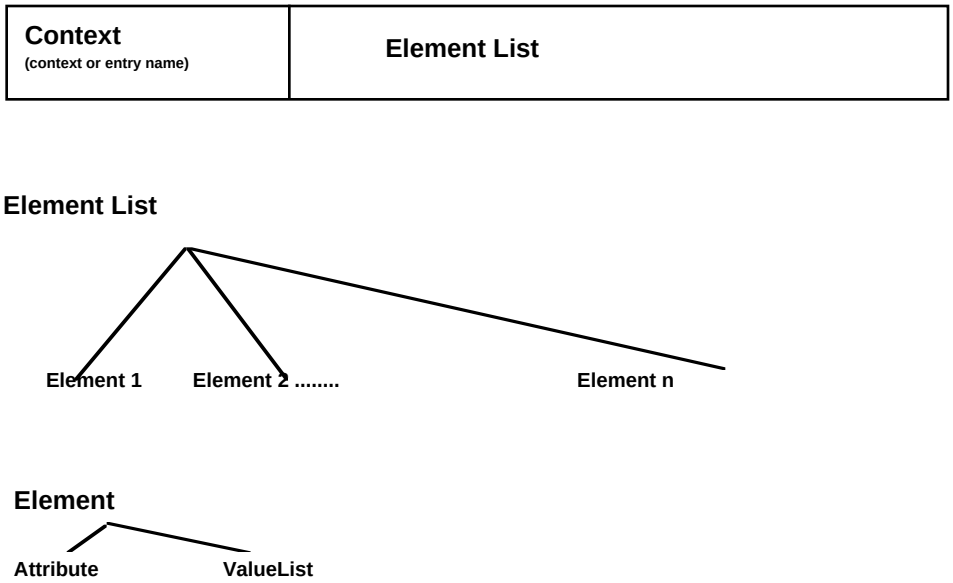

Figure 4.2.1.4-1. Naming Service - CDS Entry Structure

4.2.1.5.2.2 Interfaces

Cell Directory Service

4.2.1.5.2.3 Stimulus

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

4.2.1.5.2.4 Desired Response

The user will receive an status back after calling the add operation. It will indicate success or failure of the operation.

4.2.1.5.2.5 Participating Classes

EcDnContext, EcDnCompositeName, EcDnElement, EcDnAttribute, EcDnValue.

4.2.1.5.2.6 Pre-conditions

The entry name, its type, the attribute name, its type and value must be established.

4.2.1.5.2.7 Post-conditions

This change the state of the naming database. It adds information into the CDS.

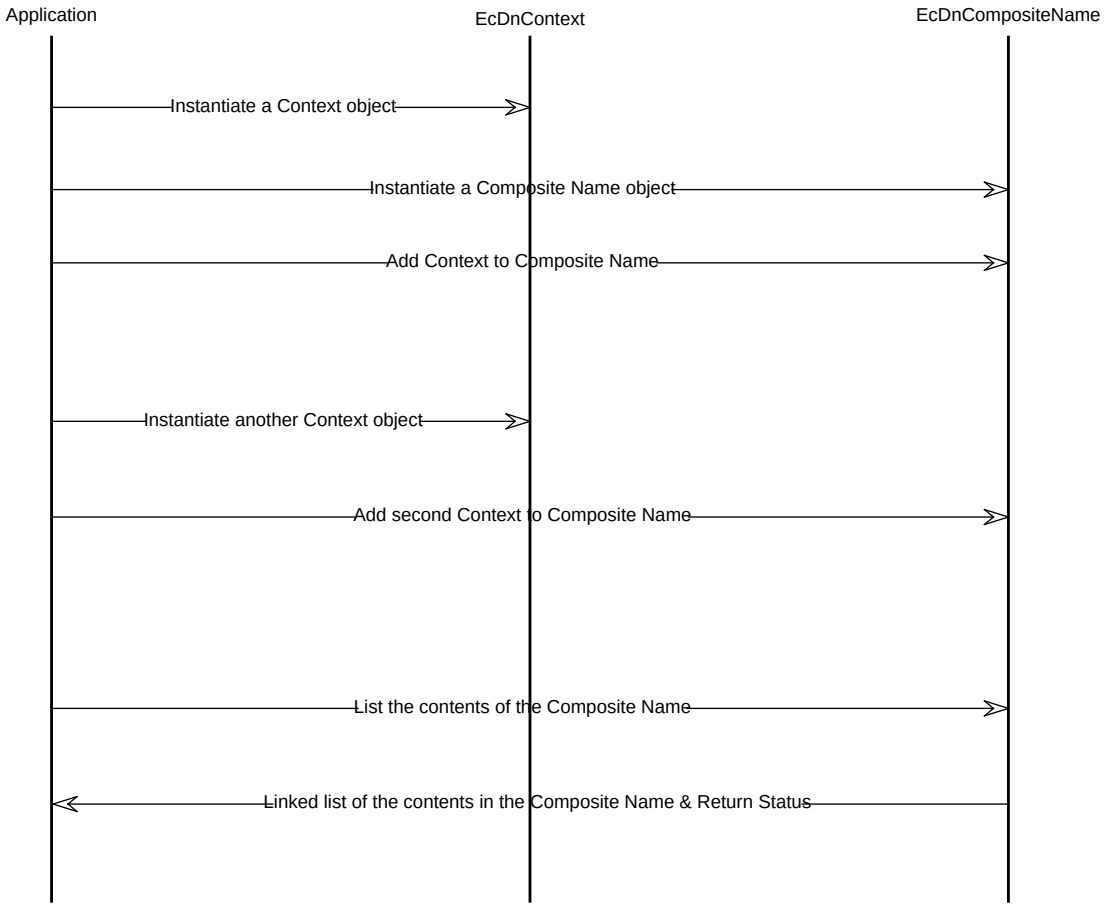

Figure 4.2.1.5-1. Naming Scenario #1

4.2.1.5.2.8 Scenario 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'.

4.2.1.5.2.9 Event Trace

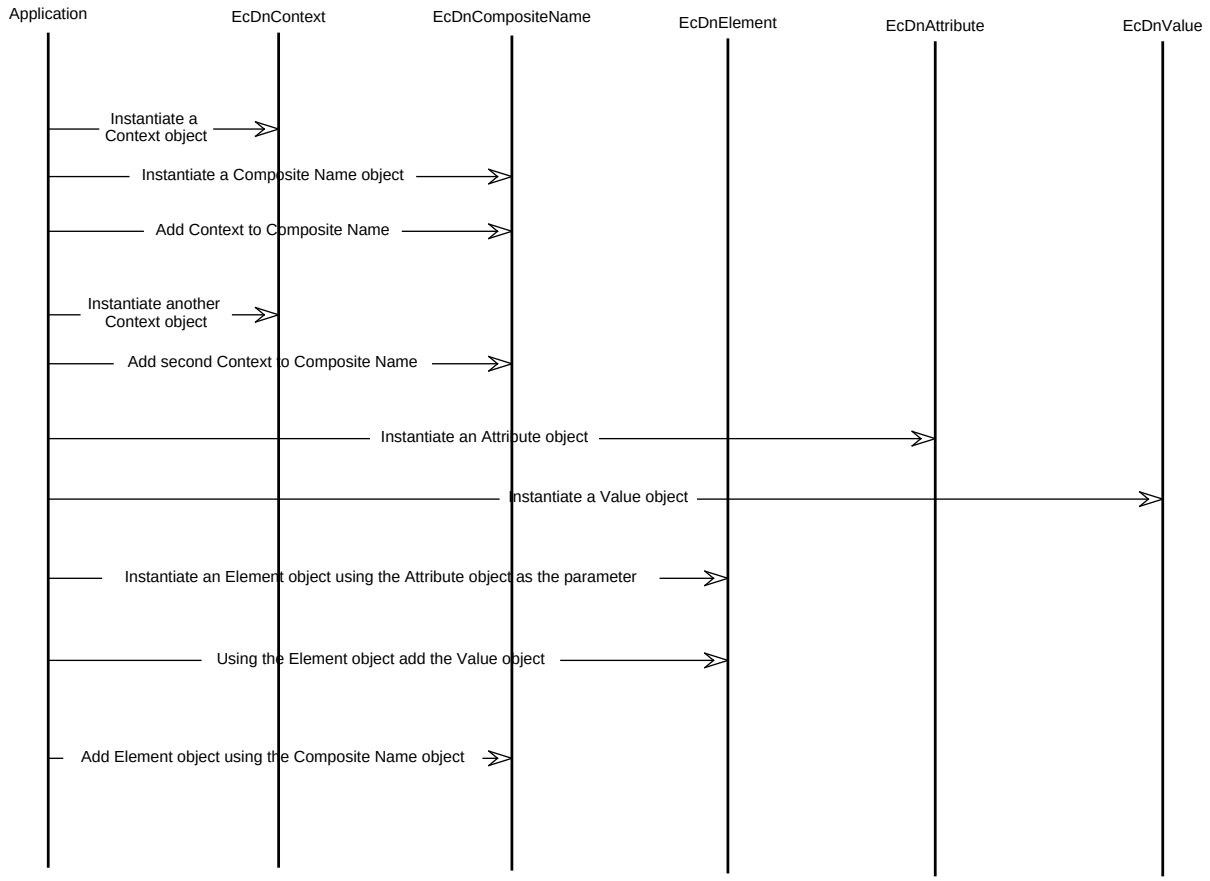

Figure 4.2.1.5-2. Naming Scenario #2

4.2.1.5.3 Scenario #3

4.2.1.5.3.1 Abstract

This scenario describes how to read an element from the Directory Service.

4.2.1.5.3.2 Interfaces

Cell Directory Service

4.2.1.5.3.3 Stimulus

The user enters a series of atomic names in order to create a composite name that is in accordance with the CDS naming conventions, and makes a request to read the contents of an entry (i.e., a file).

4.2.1.5.3.4 Desired Response

The user will get the contents of the entry (element) read, that is, the attribute list information pertaining to that element.

4.2.1.5.3.5 Participating Classes

EcDnContext, EcDnCompositeName, EcDnElement, EcDnAttribute, EcDnValue.

4.2.1.5.3.6 Pre-conditions

The composite name and entry to be read must be determined.

4.2.1.5.3.7 Post-conditions

This doesn't change the state of the naming database. It only retrieves information about the entry name. It returns a linked list of element objects (attribute/value list information).

4.2.1.5.3.8 Scenario 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.

4.2.1.5.3.9 Event Trace

4.2.1.5.4 Scenario #4

4.2.1.5.4.1 Abstract

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

4.2.1.5.4.2 Interfaces

Cell Directory Service.

4.2.1.5.4.3 Stimulus

The user enters a series of atomic names in order to create a composite name that is in accordance with the CDS naming conventions, and makes a request to list the values of an element (attribute).

4.2.1.5.4.4 Desired Response

The user will list the contents of the composite name.

4.2.1.5.4.5 Participating Classes

EcDnContext, EcDnCompositeName, EcDnElement, EcDnAttribute, EcDnValue.

4.2.1.5.4.6 Pre-conditions

The composite name and entry to be read must be determined.

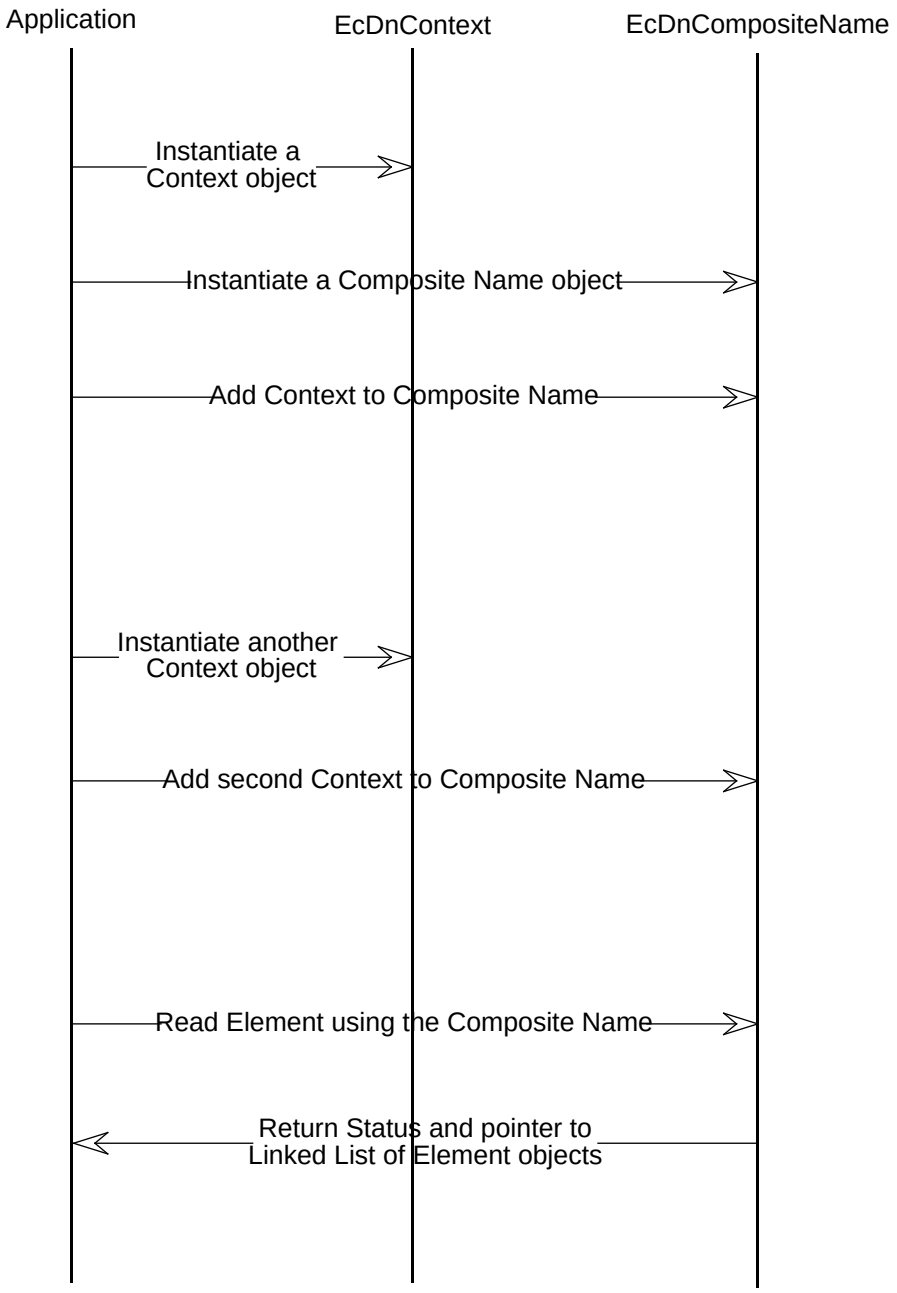

Figure 4.2.1.5-3. Naming Scenario #3

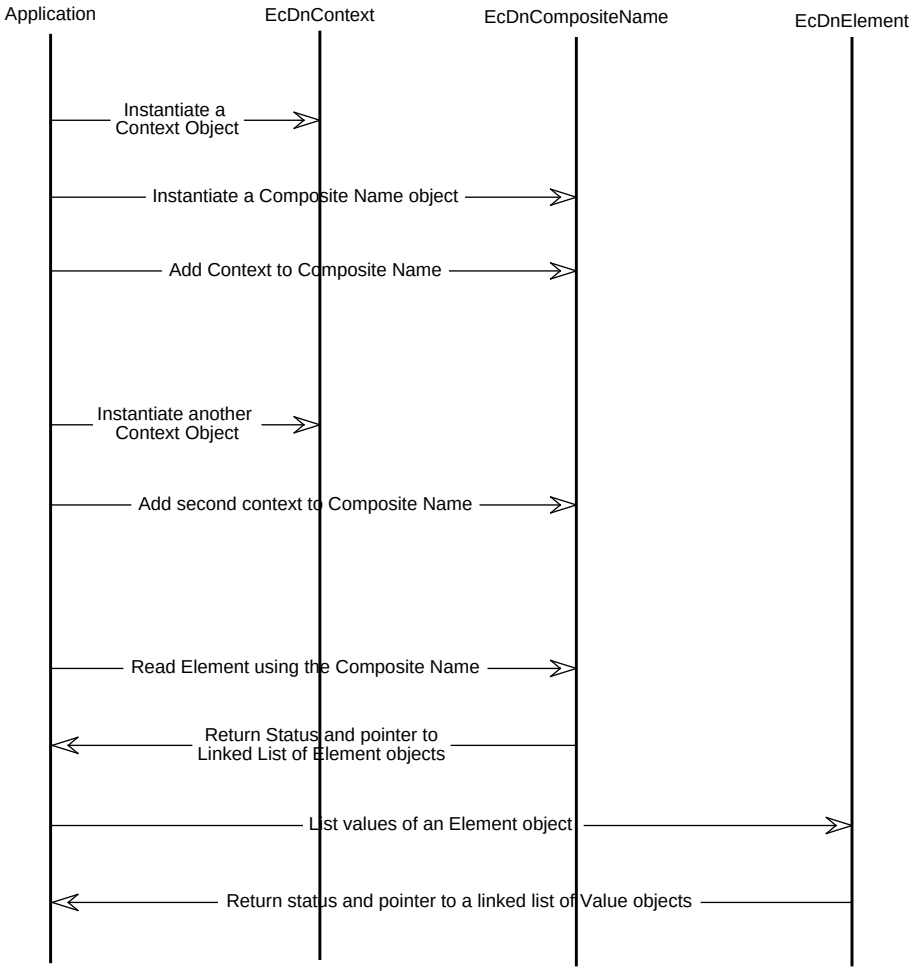

Figure 4.2.1.5-4. Naming Scenario #4

4.2.1.5.4.7 Post-conditions

This doesn't change the state of the naming database. It only retrieves information about the element. It returns a linked list of value objects pertaining to the attribute/element.

4.2.1.5.4.8 Scenario 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.

4.2.1.5.4.9 Event Trace

4.2.1.6 Implementation

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.

X/Open is currently developing code to support the XFN interface. 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.

Release B CSS will not add any additional functionalities to the directory/naming service. The XFN is currently being implemented by OSF as part of a future technology offering. We do not expect it to be available from vendors within the Release B time frame.

4.2.1.7 Service/CSCI Management and Operation

4.2.1.7.1 System Management and Strategy

DCE services are protected from a single point of failure by replicating (copying) the services and important files to additional hosts in the network. Replication ensures that critical services and data are available even after some failures. Distributed user applications can also include replication to improve availability.

4.2.1.7.2 Operator Interface

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.

4.2.1.7.3 Reports

None.

4.2.1.8 Frequently Asked Questions

The Frequently Asked Questions (FAQ) included in this section are taken from the FAQ posted in the OSF Home Page by Dave Manny and others.

1) How are directory contents replicated? What about a master and several slaves? What is a master and a slave?

   Updating the slaves is transparent. Updating is done periodically. In addition to this, the directory service provide a way for the system administrator to update replicas on demand, called skulking. Portions of the naming database can be updated.

   A master is the original copy/version of the directory contents. Usually the master is where all updates and changes are done. A slave is a copy of the directory contents. Normally, slaves only have read permission, updates are not allowed; however, some products do allow write permissions for the slaves but the slave is still a copy of the original directory contents. If a product allows write permission to slaves then there are a lot of synchronization that have to go on.

2) Where do the replicas reside?

   Replicas reside on different machines from the master, usually at a place nearer to the usage. This geographical proximity enhances performance.

3) What happens when a replica is not available?

   When a lookup is done, if a chosen replica is not available, the client internally does another lookup (a retry) with another available replica.

4) What is 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.

5) 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.

6) Is it possible for a machine to be a member of more than one DCE cell?

   Not 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.

7) Is it possible for a user in one cell to use secure services in another cell after DCE 1.1. is deployed?

   Yes. The Access Control List (ACL) contains three entry types: foreign_user, foreign_group and foreign_other which specify the permissions available to users on other cells. All that is required for intercell access, other than physical connectivity, is for the two cell's security services to be configured to know about each other.

   There is a command, the rgy_edit "cell" command, that must be run, once, by the cell administrator of the two cells that wish to communicate. After that, it's all transparent.

8) What is NSedit?

   It is a Project Pilgrim application. Naming services, in general are network services that make it possible to access resources using names rather than numbers or other cryptic forms. Some naming services also provide mechanisms through which resources can be loaded and accessed by names without knowing their physical location in the network. There are numerous types of name service implementations existing today that store information about resources. An increasing number of current applications are required to access the information from these various sources. However, each name service implementation provides a unique method for accessing similar information.

   There is no generic interface which can be used to access the various naming systems in a portable manner. Applications must use the specific interfaces provided by each naming system in order to access their respective namespaces. Additionally, some environments may not support a particular type of namespace or might provide access to the same namespace using a different interface. In such cases, users are faced with learning and using a number of different interfaces to the various namespaces.

   NSedit provides a uniform and consistent interface to the numerous name services available in an environment. Thus applications can access the various name services using the same interface. New naming systems can be easily added to the list of supported naming systems.

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 and clients may communicate in RPC or other inter-processing communications methods. 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 when the client and the server communicate via RPC or other inter-process communications methods. These mechanisms are authentication, authorization, tamper-proofing (for data integrity), encryption (for data privacy), and auditing. While authentication should always be used in every conversation between a client and a server, the mechanisms for authorization, data integrity and privacy, and auditing 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 users and different kinds of access operations. This is used to selectively grant certain principals access to some resources. Authorization is performed by an 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. An ACL manager library is provided. A noticeable feature of the authorization process for the RPC communication is that it allows the intermediary servers to operate on behalf of the initiating client while preserving both the client's and servers' identities and access control attributes across chained RPC operations.

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 the 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.

Auditing is provided to reinforce the secure communications between the client and the server as well as to protect the integrity of the distributed systems. The auditing can be incorporated into programs to support the surveillance on the fly and the auditing also allows administrators to track security-related events within trusted computing base.

While authentication is necessary, other features such as authorization, checksums, data privacy, and auditing are optional. ECS application can selectively choose to use these security features depending on their specific needs. Client and server applications using inter-process communications mechanisms other than DCE RPCs can mutually authenticate themselves with each other and communicate securely through use of the Extended Generic Security Service Application Program Interface (GSSAPI).

4.2.2.2 Context

It is assumed that all ECS services (but not necessarily all clients) run in the DCE/OODCE environment. Figure 4.2.2.2-1 depicts the different steps in the usage of this service by an application that needs to implement security. A few examples of the usage of this service within ECS are provided here. 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 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 Object Model

The object model (see Figure 4.2.2.3-1) indicates all the CSS provided security classes and the COTS provided classes and their associations with each other and with an application proxy object and application server object. Section 4.2.2.3.1 describes all the CSS provided security classes in detail about where and how they are used. For details on COTS provided security classes please refer to DCE reference manuals.

4.2.2.3.1 ECSAcl Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:True
Purpose and Description:
This class is used to access a DCE access control list. It maintains all the information about an ACL. The ECSAclDb stores Acl information, and makes it visible for reading through this interface. The ECSModifyableAcl interface is used for editing ACLs. A ECSAcl is a read-only object. In order to change a ECSAcl, it must be either replaced completely with another ECSAcl, or converted into a ECSModifyableAcl in order to be edited through an application interface.

Attributes:

None

Operations:

ECSAcl - This is the default constructor. It creates a ECSAcl object.
Arguments:
Return Type:Void
Privilege:Public

GetAccess - This operation returns the set of permissions that can be granted to the current client by the ECSAcl.
Arguments:
Return Type:DCESchemaBitset
Privilege:Public

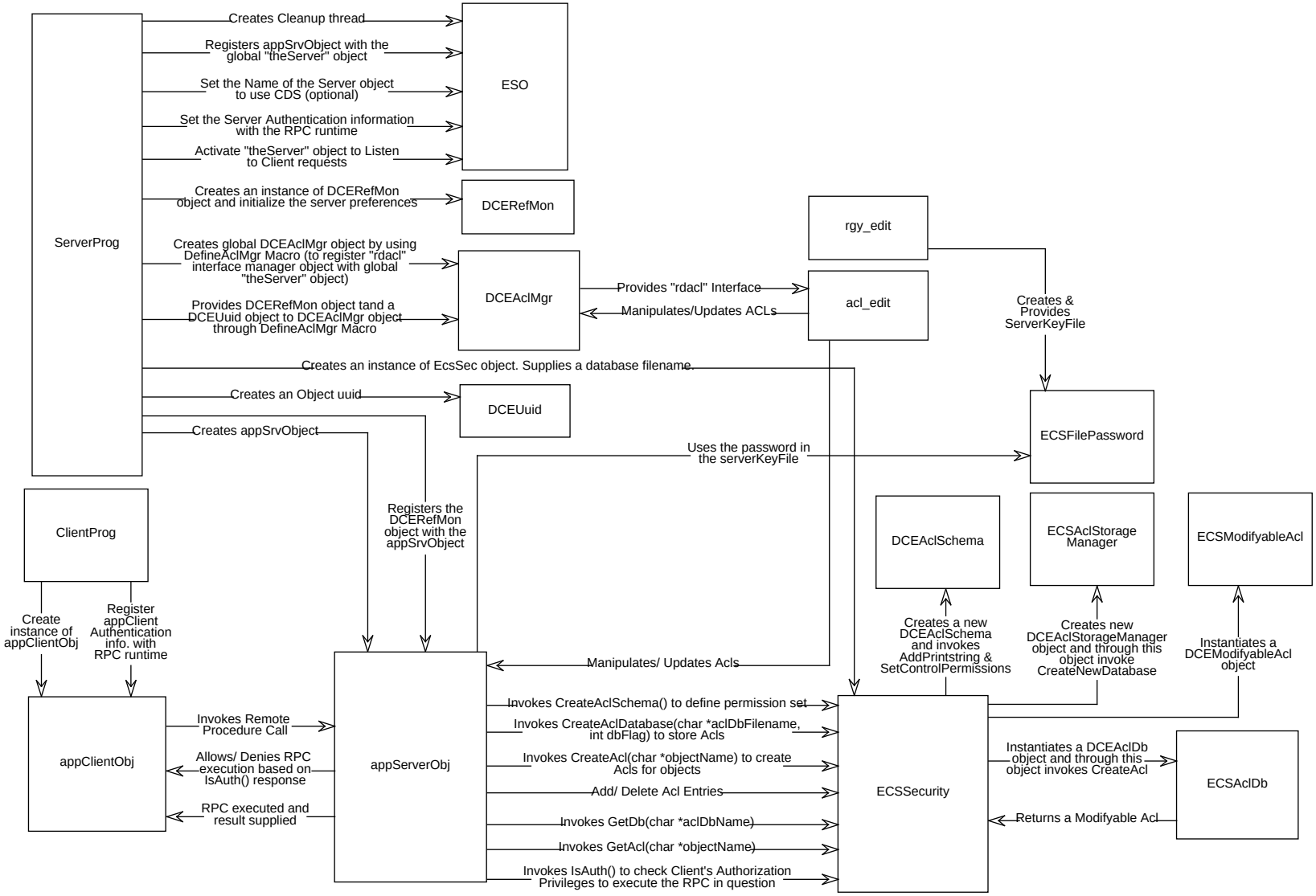

Figure 4.2.2.2-1. Security Service Event Flow

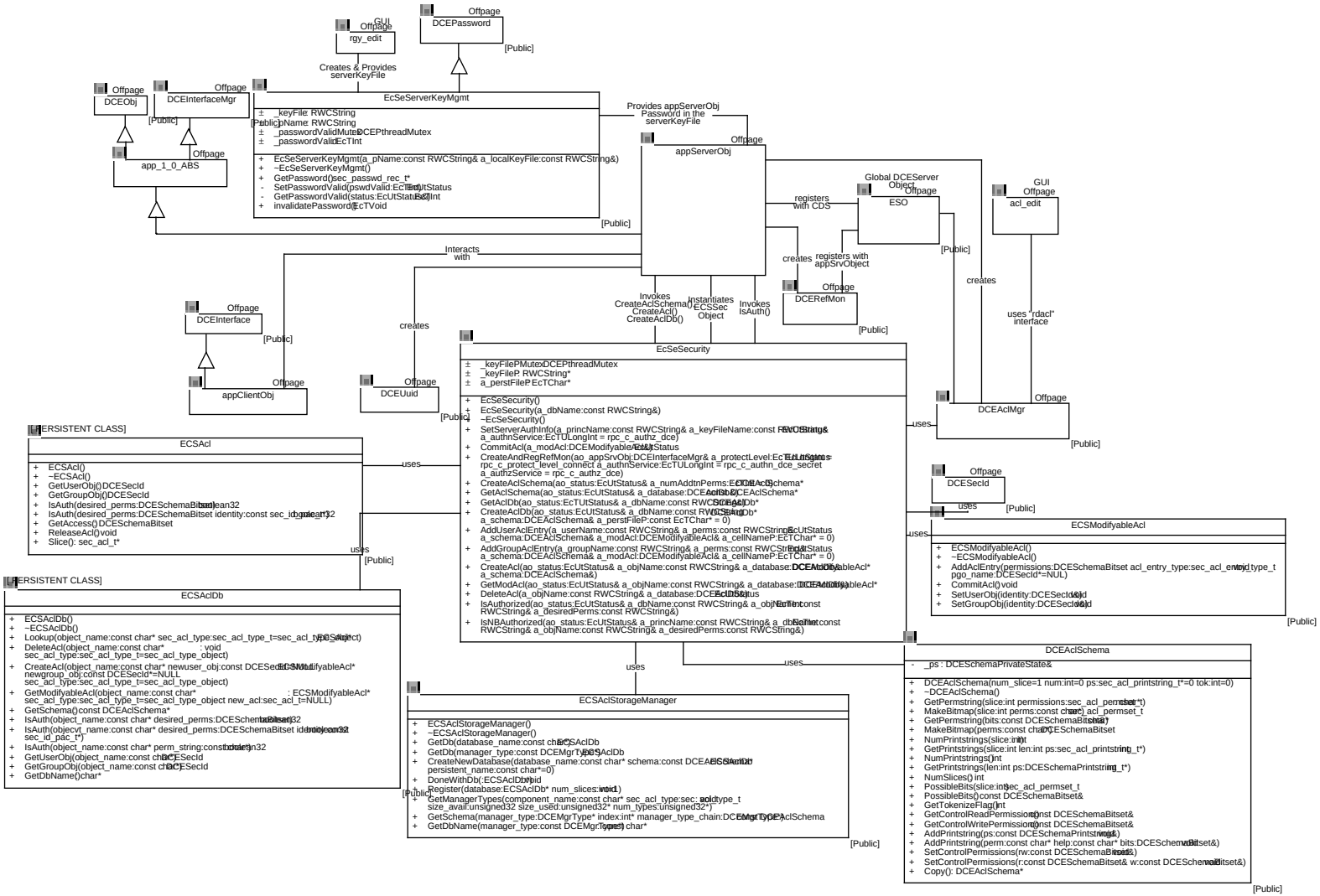

Figure 4.2.2.3-1. Security Service Object Model

GetGroupObj - This operation will return the group owner identity. The same explanation applies as for GetUserObj().
Arguments:
Return Type:DCESecId
Privilege:Public

GetUserObj - This operation will return the user owner-identity associated with the ECSAcl. It is a convenience routine because the information might actually be stored in the database, in which case the ECSAcl needs to get the information from the database.
Arguments:
Return Type:DCESecId
Privilege:Public

IsAuth - This operation is the most important member function that ECSAcl provides. It returns TRUE if the ECSAcl will grant the desired permissions to the client. It returns FALSE if all the desired permissions can not be granted.
Arguments:desired_perms:DCESchemaBitset
Return Type:boolean32
Privilege:Public

IsAuth - This operation checks the authorization rights of a third party whose identity is passed to the function as the second argument.
Arguments:desired_perms:DCESchemaBitset identity:const sec_id_pac_t*
Return Type:boolean32
Privilege:Public

ReleaseAcl - This operation advises the system that the ECSAcl is no longer needed. In the implementation to be provided, a call to this function unlocks the ECSAcl so that other clients may use it.
Arguments:
Return Type:void
Privilege:Public

Slice - This operation will return the DCE data structure sec_acl_t. It extracts just one slice of permissions since that is all that a sec_acl_t will hold. This conversion from an ECSAcl object to a sec_acl is needed by the 'rdacl' interface.
Arguments:
Return Type:sec_acl_t*
Privilege:Public

~ECSAcl - This is the destructor. It deletes a ECSAcl object.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The ECSAcl class has associations with the following classes:
Class: EcSeSecurity uses

4.2.2.3.2 ECSAclDb Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:True
Purpose and Description:
This class defines the interface to the ACL database. The ACL databases store and retrieve all ACLs based on an object name. ACLs can be organized and retrieved in different ways such as by ACL type (object, default object, or default container), and according to the permission slice. Each database is associated with only one DCEAclSchema that may contain multiple slices. The ECSAclDb implementation must define the ACL locking protocol and provide the mechanism to use it.

Attributes:

None

Operations:

CreateAcl - This operation will create an ECSModifyableAcl object that is brand new; that is, there was no ECSAcl in this database corresponding to the obejct name and ECSAcl type supplied. It creates a blank template that can be filled in and written back to the database using the ECSModifyableAcl member functions. The owner and the group identities may be specified here or may be filled in later by the member functions of ECSModifyableAcl.
Arguments:object_name:const char* newuser_obj:const DCESecId=NULL newgroup_obj:const DCESecId*=NULL sec_acl_type:sec_acl_type_t=sec_acl_type_object
Return Type:ECSModifyableAcl*
Privilege:Public

DeleteAcl - This operation will delete an ECSAcl from the database given the object name and an optional ACL type.
Arguments:object_name:const char* sec_acl_type:sec_acl_type_t=sec_acl_type_object
Return Type:void
Privilege:Public

ECSAclDb - This is the default constructor.
Arguments:

Return Type:Void
Privilege:Public

GetDbName - This operation will return the name of the ECSAclDb database in character string format.
Arguments:
Return Type:char*
Privilege:Public

GetGroupObj - This operation will return a DCESecId object that represents the group owner of the object; that is, the identity of the group that will match on the group_obj ACL entry.
Arguments:object_name:const char*
Return Type:DCESecId
Privilege:Public

GetModifyableAcl - This operation should be called when users want to edit an Acl in an application interface rather than through the 'rdacl' interface. It creates a ECSModifyableAcl object containing the current existing Acl information for the specified object in the database. The changes made to the ECSModifyableAcl are not used in authorization decisions until they are explicitly committed back to the database.
Arguments:object_name:const char* sec_acl_type:sec_acl_type_t=sec_acl_type_object new_acl:sec_acl_t=NULL
Return Type:ECSModifyableAcl*
Privilege:Public

GetSchema - This operation will return a pointer to the schema object that describes the permission bits managed by this database.
Arguments:
Return Type:const DCEAclSchema*
Privilege:Public

GetUserObj - This operation returns a DCESecId object that represents the user owner of the object; that is, the identity of the principal that will match on the user_obj ACL entry.
Arguments:object_name:const char*
Return Type:DCESecId
Privilege:Public

IsAuth - This operation make an authorization decision. It returns a TRUE if the caller has the desired_perms and a FALSE otherwise. The caller's information is obtained from within the implementation of this member function supplied by the DCEClientInfo object that is obtained when the server is invoked either through the application interface or thropugh the 'rdacl' interface.
Arguments:object_name:const char* desired_perms:DCESchemaBitset
Return Type:boolean32

Privilege:Public

IsAuth - This operation is similar to the first form but accepts a string representation of the permission set desired instead of a DCESchemaBitset.
Arguments:objecvt_name:const char* desired_perms:DCESchemaBitset identity:const sec_id_pac_t*
Return Type:boolean32
Privilege:Public

IsAuth - This is similar to the first and second form except that it takes a DCE PAC structure to identify a third party. If the requestor and that third party have the desired permissions, it returns a TRUE. This form is needed to implement rdacl_get_access_on_behalf.
Arguments:object_name:const char* perm_string:const char*
Return Type:boolean32
Privilege:Public

Lookup - This operation will lookup an ECSAcl in the database based on the object name and the type of ACL. It returns a pointer to an ECSAcl. A ECSAcl is read-only and none of the member functions can modify its state.
Arguments:object_name:const char* sec_acl_type:sec_acl_type_t=sec_acl_type_object
Return Type:ECSAcl*
Privilege:Public

~ECSAclDb - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The ECSAclDb class has associations with the following classes:
Class: EcSeSecurity uses

4.2.2.3.3 ECSAclStorageManager Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class supports multiple ACL databases. It manages the ACL databases being used by a server, providing registration and search services for these databases. It provides manager type and schema information as needed by the rdacl interface. It also provides functionality for creating a new ECSAclDb.

Attributes:

None

Operations:

CreateNewDatabase - This operation creates a new DCEAclDb object with the given name and associated with the given schema. The third optional parameter allows the database implementor to provide a constructor that uses the full pathname of a persistent datastore.
Arguments:database_name:const char* schema:const DCEAclSchema* persistent_name:const char*=0
Return Type:ECSAclDb
Privilege:Public

DoneWithDb - This operation will indicate when done with the specified database.
Arguments::ECSAclDb*
Return Type:void
Privilege:Public

ECSAclStorageManager - This is the class default constructor. It creates an ECSAclStorageManager object.
Arguments:
Return Type:Void
Privilege:Public

GetDb - This operation provides a database handle from the DCEAclStorageMgr by providing the database name.
Arguments:database_name:const char*
Return Type:ECSAclDb
Privilege:Public

GetDb - This operation provides a database handle from the DCEAclStorageMgr by providing a manager type.
Arguments:manager_type:const DCEMgrType*
Return Type:ECSAclDb
Privilege:Public

GetDbName - This operation returns the name of the database corresponding to the manager type passed in. This is used in constructing the print string for the manager type required by rdacl_get_printstrings.
Arguments:manager_type:const DCEMgrType*
Return Type:const char*
Privilege:Public

GetManagerTypes - This operation will return the array of DCEMgrType corresponding to the databases that contain the ACLs protecting the object identified by the first parameter. It supports the implementation of rdacl_get_manager_types and rdacl_get_manager_types_semantics.
Arguments:component_name:const char* sec_acl_type:sec_acl_type_t size_avail:unsigned32 size_used:unsigned32* num_types:unsigned32*
Return Type:void
Privilege:Public

GetSchema - This operation is convenient for implementing the 'rdacl' interface, and can be ignored by most users. It returns a pointer to a DCEAclSchema object for the database identified by the first parameter, which is manager type. In this case, the manager type identifies a particular slice of permissions.
Arguments:manager_type:DCEMgrType* index:int* manager_type_chain:DCEMgrType*
Return Type:const DCEAclSchema
Privilege:Public

Register - This operation allows the server developer to construct an ECSAclDb and then to register it directly with the ECSAclStorageManager object.
Arguments:database:ECSAclDb* num_slices:int=1
Return Type:void
Privilege:Public

~ECSAclStorageManager - This is the class destructor. It frees the memory occupied by the ECSAclStorageManager object.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The ECSAclStorageManager class has associations with the following classes:
Class: EcSeSecurity uses

4.2.2.3.4 ECSModifyableAcl Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This is a temporary copy of ECSAcl that can be used for editing directly by the server or through an application-defined interface. Users of the 'rdacl' interface do not use ECSModifyableAcl directly, but rather ECSModifyableAcl provides an alternative path for

editing ACLs. The contents of a ECSModifyableAcl are not seen when authorization decisions are made. The changes don't take effect until the ECSModifyableAcl is committed back to the database, at which time the ECSModifyableAcl itself no longer exists.

Attributes:

None

Operations:

AddAclEntry - This operation can be used to add new ACL entries, once a ECSModifyableAcl object has been created. It takes permissions, the acl_entry_type, and the optional pgo_name identity corresponding to the ACL entry.
Arguments:permissions:DCESchemaBitset acl_entry_type:sec_acl_entry_type_t pgo_name:DCESecId*=NUL
Return Type:void
Privilege:Public

CommitAcl - This operation will write the ECSModifyableAcl contents into the database and deletes the ECSModifyableAcl. When CommitAcl() completes, authorization decisions will take the new ACL into account.
Arguments:
Return Type:void
Privilege:Public

ECSModifyableAcl - This is the class default constructor.
Arguments:
Return Type:Void
Privilege:Public

SetGroupObj - This operation will allow the caller to change the identity of the group owner of the object protected by this ACL.
Arguments:identity:DCESecId&
Return Type:void
Privilege:Public

SetUserObj - This operation allows the caller to change the identity of the user owner of the object protected by this ACL.
Arguments:identity:DCESecId&
Return Type:void
Privilege:Public

~ECSModifyableAcl - This is the class destructor.

Arguments:
Return Type:Void
Privilege:Public

Associations:

The ECSModifyableAcl class has associations with the following classes:
Class: EcSeSecurity uses

4.2.2.3.5 EcSeSecurity Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
This class is the security class. When it is required to implement security service in any application, it is required by the application developer to instantiate an EcSeSecurity object in the serverMain program. This class makes use of the COTS provided classes such as DCEAclSchema, DCESchemaBitset, DCESecId, etc. and also CSS customized classes such as ECSAcl, ECSAclDb, ECSModifyableAcl and ECSAclStorageManager. The ECSAclStorageManager object is per DCE server. Applications can access it through the global reference named ECSAclStorageMgr. ECSAclStorageManager class maintains a table of known ACL databases. Each database can contain ACLs for more than one object. ECSSecurity makes use of ECSAclStorageManager to CreateAclDatabase with the persistent storage feature and to GetDatabaseName. Similarly ECSSecurity uses DCEAclSchema to SetControlPermissions and AddPrintstrings to ACLS, uses ECSAclDb to CreateAcls supplying the name of the object for which the ACL is to be created, to GetAcl and to perform the authorization check on client's privileges to access any resource through IsAuth function.

Attributes:

_keyFileP - This attribute identifies a file that stores the password of a dce principal (ex: non-interactive principals such as servers).
Data Type:RWCString*
Privilege:Protected
Default Value:

_keyFilePMutex - This attribute represents the pthread Mutex for l_keyFileP class variable.
Data Type:DCEPthreadMutex
Privilege:Protected
Default Value:

a_perstFileP - This attribute identifies a file which is used for persistent storage of Acls.

Data Type:EcTChar*
Privilege:Protected
Default Value:

Operations:

AddGroupAclEntry - This member function will add acl entry for a given group in the given ACL.
Arguments:a_groupName:const RWCString& a_perms:const RWCString& a_schema:DCEAclSchema& a_modAcl:DCEModifyableAcl& a_cellNameP:EcTChar* = 0
Return Type:EcUtStatus
Privilege:Public

AddUserAclEntry - This member function will add an Acl entry for a given user in the given ACL.
Arguments:a_userName:const RWCString& a_perms:const RWCString& a_schema:DCEAclSchema& a_modAcl:DCEModifyableAcl& a_cellNameP:EcTChar* = 0
Return Type:EcUtStatus
Privilege:Public

CommitAcl - This method will commit any changes done to the ACLs.
Arguments:a_modAcl:DCEModifyableAcl&
Return Type:EcUtStatus
Privilege:Public

CreateAcl - This method will create an ACL for a given resource/object/service name.
Arguments:ao_status:EcUtStatus& a_objName:const RWCString& a_database:DCEAclDb& a_schema:DCEAclSchema&
Return Type:DCEModifyableAcl*
Privilege:Public

CreateAclDb - This function will create a database for given ACL, where the ACLs can be stored.
Arguments:ao_status:EcUtStatus& a_dbName:const RWCString a_schema:DCEAclSchema& a_perstFileP:const EcTChar* = 0
Return Type:DCEAclDb*
Privilege:Public

CreateAclSchema - This method will create a schema with default allowable permissions being read, write, execute, insert, delete, test, acl-control (i.e. rwxidtc) for the Acls. It also provides an option of 25 additional permissions A-Y which can be used for any application specific needs.

Arguments:ao_status:EcUtStatus& a_numAddtnPerms:EcTInt = 0
Return Type:DCEAclSchema*
Privilege:Public

CreateAndRegRefMon - This function will create an instance of the DCEStdRefMon object and register with the manager object.
Arguments:ao_appSrvObj:DCEInterfaceMgr& a_protectLevel:EcTULongInt = rpc_c_protect_level_connect a_authnService:EcTULongInt = rpc_c_authn_dce_secret a_authzService = rpc_c_authz_dce
Return Type:EcUtStatus
Privilege:Public

DeleteAcl - This function will delete an Acl given the object/resource name.
Arguments:a_objName:const RWCString& a_database:DCEAclDb&
Return Type:EcUtStatus
Privilege:Public

EcSeSecurity - This is the default constructor.
Arguments:
Return Type:Void
Privilege:Public

EcSeSecurity - This constructor takes database filename as an input parameter. When this is invoked, if a database with the supplied filename already exists in DCEAclStorageMgr(DCEAclStorageMgr maintains a list of databases, which is initially empty), the file is used for reading, else a new database with the given name is created.
Arguments:a_dbName:const RWCString&
Return Type:Void
Privilege:Public

GetAclDb - This function will get the Acl Database where the Acl in question is stored, which is already created by the CreateAclDb() method.
Arguments:ao_status:EcTUtStatus& a_dbName:const RWCString&
Return Type:DCEAclDb*
Privilege:Public

GetAclSchema - This function will get the AclSchema for the Acl in question, which is already created by CreateAclSchema() method.
Arguments:ao_status:EcUtStatus& a_database:DCEAclDb&
Return Type:const DCEAclSchema*
Privilege:Public

GetModAcl - This method will get the DCEModifyableAcl object for a given object/ resource.
Arguments:ao_status:EcUtStatus& a_objName:const RWCString&

a_database:DCEAclDb&
Return Type:DCEModifyableAcl*
Privilege:Public

IsAuthorized - This is a very important member function. It checks the authorization privileges of a client (based on client's principal name OR the group to which he belongs to) requesting operation/access for a given resource/service. It returns a boolean value indicating the denial/granting of the requested permission/access.
Arguments:ao_status:EcUtStatus& a_dbName:const RWCString& a_objName:const RWCString& a_desiredPerms:const RWCString&
Return Type:EcTInt
Privilege:Public

IsNBAuthorized - This method is the most important method in the EcSeSecurity class. This method will perform authorization check on the caller's privileges (obtained from the caller's principal name) and comparing it against the ACL maintained for the resource in question) before permitting him/her to perform any kind of access to any of his requested service/resource
Arguments:ao_status:EcUtStatus& a_princName:const RWCString& a_dbName:const RWCString& a_objName:const RWCString& a_desiredPerms:const RWCString&
Return Type:EcTInt
Privilege:Public

SetServerAuthInfo - This member function will set the server authentication information(i.e. authentication protocol, authorization protocol, protection levels) with the rpc runtime.
Arguments:a_princName:const RWCString& a_keyFileName:const RWCString& a_authnService:EcTULongInt = rpc_c_authz_dce
Return Type:EcUtStatus
Privilege:Public

~EcSeSecurity - This is the destructor. It deletes a EcSeSecurity object.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSeSecurity class has associations with the following classes:
Class: appServerObj InstantiatesECSSecObject
Class: appServerObj InvokesCreateAclSchema(),CreateAcl()CreateAclDb()
Class: appServerObj InvokesIsAuth()
Class: DCEAclMgr uses
Class: DCEAclSchema uses
Class: DCESecId uses

Class: ECSAcl uses
Class: ECSAclDb uses
Class: ECSAclStorageManager uses
Class: ECSModifyableAcl uses

4.2.2.3.6 EcSeServerKeyMgmt Class

Parent Class:DCEPassword
Public:Yes
Distributed Object:No
Purpose and Description:
This class is the concrete implementation of the abstract DCEPassword class using a file as the means of retrieving the secret key. This class provides a consistent way for accessing the password data for a particular security principal (non interactive principal - server).

Attributes:

_keyFile - This attribute represents the server keytab file where the server's password is encoded.
Data Type:RWCString
Privilege:Protected
Default Value:

_pName - This is the principal name whose password is stored in the keytab file.
Data Type:RWCString
Privilege:Protected
Default Value:

_passwordValid - This attribute represents a flag which indicates whether a password is valid or invalid.
Data Type:EcTInt
Privilege:Protected
Default Value:

_passwordValidMutex - This attribute represents the pthread mutex for the _passwordValid data member.
Data Type:DCEPthreadMutex
Privilege:Protected
Default Value:

Operations:

EcSeServerKeyMgmt - This is the class constructor. It makes a copy of the supplied principalName and keyFileName into respective class variables.
Arguments:a_pName:const RWCString& a_localKeyFile:const RWCString&
Return Type:Void

Privilege:Public

GetPassword - This operation is the implementation of the base class pure virtual making a dce call to retrieve the secret key of a principal from the _keyFile.
Arguments:
Return Type:sec_passwd_rec_t*
Privilege:Public

GetPasswordValid - This method will get the value of the password validity(0 or 1).
Arguments:status:EcUtStatus&
Return Type:EcTInt
Privilege:Private

SetPasswordValid - This member function will set the password validity to 0 or 1.
Arguments:pswdValid:EcTInt
Return Type:EcUtStatus
Privilege:Private

invalidatePassword - Renders a password not valid.
Arguments:
Return Type:EcTVoid
Privilege:Public

~EcSeServerKeyMgmt - This is the class destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSeServerKeyMgmt class has associations with the following classes:
Class: rgy_edit Creates&ProvidesserverKeyFile
Class: appServerObj ProvidesappServerObjPasswordintheserverKeyFile

4.2.2.3.7 DCEAclMgr Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class registers a 'rdacl' interface manager object with the global DCEServer object.

Attributes:

None

Operations:

None

Associations:

The DCEAclMgr class has associations with the following classes:
Class: ESO
Class: appServerObj creates
Class: acl_edit uses"rdacl"interface
Class: EcSeSecurity uses

4.2.2.3.8 DCEAclSchema Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class defines the permission bits that are available and provides a printable form of each bit and an explanatory string.

Attributes:

_ps - This attribute represents all the private state in it.
Data Type:DCESchemaPrivateState&
Privilege:Private
Default Value:

Operations:

AddPrintstring - This operation will add new print strings to a schema.
Arguments:ps:const DCESchemaPrintstring&
Return Type:void
Privilege:Public

AddPrintstring - This operation will add new print strings to a schema.
Arguments:perm:const char* help:const char* bits:DCESchemaBitset&
Return Type:void
Privilege:Public

Copy - This operation creates a copy of the DCEAclSchema.

Arguments:
Return Type:DCEAclSchema*
Privilege:Public

DCEAclSchema - This is the class constructor. It creates a DCEAclSchema object.
Arguments:num_slice=1 num:int=0 ps:sec_acl_printstring_t*=0 tok:int=0
Return Type:Void
Privilege:Public

GetControlReadPermission - This operation will retrieve the definition of the READ control permission for the schema.
Arguments:
Return Type:const DCESchemaBitset&
Privilege:Public

GetControlWritePermission - This operation will retrieve the definition of control permissions in the schema that a user must have to access the ACL information.
Arguments:
Return Type:const DCESchemaBitset&
Privilege:Public

GetPermstring - This operation returns a character string representation of permissions, either for a slice or for an entire permissions set.
Arguments:slice:int permissions:sec_acl_permset_t
Return Type:char*
Privilege:Public

GetPermstring - This operation will return a character string representation of permissions, either for a slice or for an entire permissions set.
Arguments:bits:const DCESchemaBitset&
Return Type:char*
Privilege:Public

GetPrintstrings - This operation will fill an array ps with print string values. These are the print strings for a particular value or for a particualar schema. len is the number of elements in the ps array.
Arguments:slice:int len:int ps:sec_acl_printstring_t*
Return Type:int
Privilege:Public

GetPrintstrings - This operation will fill an array with print string values. These are either the print strings for a particular slice, or for the entire schema. len is the number of elements in the array.
Arguments:len:int ps:DCESchemaPrintstring_t*
Return Type:int

Privilege:Public

GetTokenizeFlag - This operation will return the tokenized flag.
Arguments:
Return Type:int
Privilege:Public

MakeBitmap - This operation will return a bit string which corresponds to the character form of permissions passed in.
Arguments:slice:int perms:const char*
Return Type:sec_acl_permset_t
Privilege:Public

MakeBitmap - This operation returns a bit string which corresponds to the character form of permissions passed in.
Arguments:perms:const char*
Return Type:DCESchemaBitset
Privilege:Public

NumPrintstrings - This operation will return the number of print strings in the schema, either for a particular slice or for the entire schema.
Arguments:slice:int
Return Type:int
Privilege:Public

NumPrintstrings - This operation will return the number of print strings in the schema, either for a particular slice or for the entire schema.
Arguments:
Return Type:int
Privilege:Public

NumSlices - This operation will return the number of permission slices.
Arguments:
Return Type:int
Privilege:Public

PossibleBits - This operation will return the union of the permission bits that have been defined for this print string.
Arguments:slice:int
Return Type:sec_acl_permset_t
Privilege:Public

PossibleBits - This operation will return the union of the permission bits that have been defined for this print string.
Arguments:

Return Type:const DCESchemaBitset&
Privilege:Public

SetControlPermissions - This operation will set the control bit that determines whether a user has access to the ACL structure. It indicates the permission bit selected for write access.
Arguments:rw:const DCESchemaBitset&
Return Type:void
Privilege:Public

SetControlPermissions - This operation will set the control bit that determines whether a user has access to the ACL structure. It grants read access to view ACLs seperately from granting access to modify the ACLs.
Arguments:r:const DCESchemaBitset& w:const DCESchemaBitset&
Return Type:void
Privilege:Public

~DCEAclSchema - This is the class destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The DCEAclSchema class has associations with the following classes:
Class: EcSeSecurity uses

4.2.2.3.9 DCEPassword Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class supports implementation of password storage and access.

Attributes:

None

Operations:

None

Associations:

The DCEPassword class has associations with the following classes:
None

4.2.2.3.10 DCERefMon Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:False
Purpose and Description:
This class provides an abstraction of a reference monitor that controls the client object's access to a manager object.

Attributes:

None

Operations:

None

Associations:

The DCERefMon class has associations with the following classes:
Class: appServerObj creates
Class: ESO registerswithappSrvObject

4.2.2.3.11 DCESecId Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This utility class encapsulates the sec_id_t data type of DCE.

Attributes:

None

Operations:

None

Associations:

The DCESecId class has associations with the following classes:
Class: EcSeSecurity uses

4.2.2.3.12 DCEUuid Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This utility class encapsulates the DCE data type uuid_t. It provides a flexible way of dealing with the various forms of representing a UUID.

Attributes:

None

Operations:

None

Associations:

The DCEUuid class has associations with the following classes:
Class: appServerObj creates

4.2.2.3.13 acl_edit Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
This is a GUI utility that is used to create and maintain access control lists (acls).

Attributes:

None

Operations:

None

Associations:

The acl_edit class has associations with the following classes:
Class: DCEAclMgr uses"rdacl"interface

4.2.2.3.14 appClientObj Class

Parent Class:DCEInterface
Public:No
Distributed Object:No
Purpose and Description:
This is an application client/proxy object.

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The appClientObj class has associations with the following classes:
Class: appServerObj Interactswith

4.2.2.3.15 appServerObj Class

Parent Class:app_1_0_ABS
Public:No
Distributed Object:No
Purpose and Description:
This is an application server object.

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The appServerObj class has associations with the following classes:
Class: EcSeSecurity InstantiatesECSSecObject
Class: appClientObj Interactswith
Class: EcSeSecurity InvokesCreateAclSchema(),CreateAcl()CreateAclDb()
Class: EcSeSecurity InvokesIsAuth()
Class: EcSeServerKeyMgmt ProvidesappServerObjPasswordintheserverKeyFile
Class: DCEAclMgr creates
Class: DCERefMon creates
Class: DCEUuid creates
Class: ESO registerswithCDS

4.2.2.3.16 app_1_0_ABS Class

Parent Class:DCEInterfaceMgr
Parent Class:DCEObj
Public:No
Distributed Object:No
Purpose and Description:
This is the application Abstract class generated by the application IDL.

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The app_1_0_ABS class has associations with the following classes:
None

4.2.2.3.17 rgy_edit Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Persistent Class:False
Purpose and Description:

This is a GUI utility used to crete and maintain user accounts.

Attributes:

None

Operations:

None

Associations:

The rgy_edit class has associations with the following classes:
Class: EcSeServerKeyMgmt Creates&ProvidesserverKeyFile

4.2.2.4 Functional Model

A brief description of all the CSS customized security objects as well as the COTS provided classes used by the CSS security objects is provided in the Table 4.2.2.4-1 below. For more information on any of the COTS provided objects, please refer to the OODCE reference manuals.

4.2.2.5 Dynamic Model Scenarios

4.2.2.5.1 Scenario #1

Table 4.2.2.4-1. Security Object Responsibility Matrix (1 of 2)

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.

Table 4.2.2.4-1. Security Object Responsibility Matrix (2 of 2)

Class	Description
DCEAclMgr	This class registers a rdacl interface manager object with the global DCEServer(ESO) object. DefineAclMgr macro is used to construct this class.
DCESecId	This is a utility class that encapsulates the sec_id_t data type.
DCEAclSchema	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.
EcSeServerKeyMgmt	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.
EcSeSecurity	Provides higher level functionality to authenticate principals accessing resources. These include create/update/delete ACLs, define permissions sets, persistence to the ACL database.

4.2.2.5.1.1 Abstract

This is the entire security scenario which traces the interactions of the application (which implements security) client and server objects with the global server object ESO, the EcSeSecurity, EcSeServerKeyMgmt, etc. The following occurs in this scenario:

- Setting client and server authentication information with RPC runtime.
- Initialization aspects such as creating instances of different objects and registering them.
- Creation of ACLs, AclSchema and AclDatabase.
- Invoking of IsAuth to check client's Authorization Privileges.

4.2.2.5.1.2 Interfaces

CSS's EcSeSecurity and EcSeServerKeyMgmt.

4.2.2.5.1.3 Stimulus

To create and maintain access control lists(ACLs) for resources, and with the help of these ACLs perform security check to find out if a caller is authorized to access specified resource.

4.2.2.5.1.4 Desired Response

The database server should have modifiable ACL/s for object/s which consists of ACL entries that allow a permission set defined by ACL schema, an ACL database to store the ACL created. Functionality to check caller's privilege to access any desired resource.

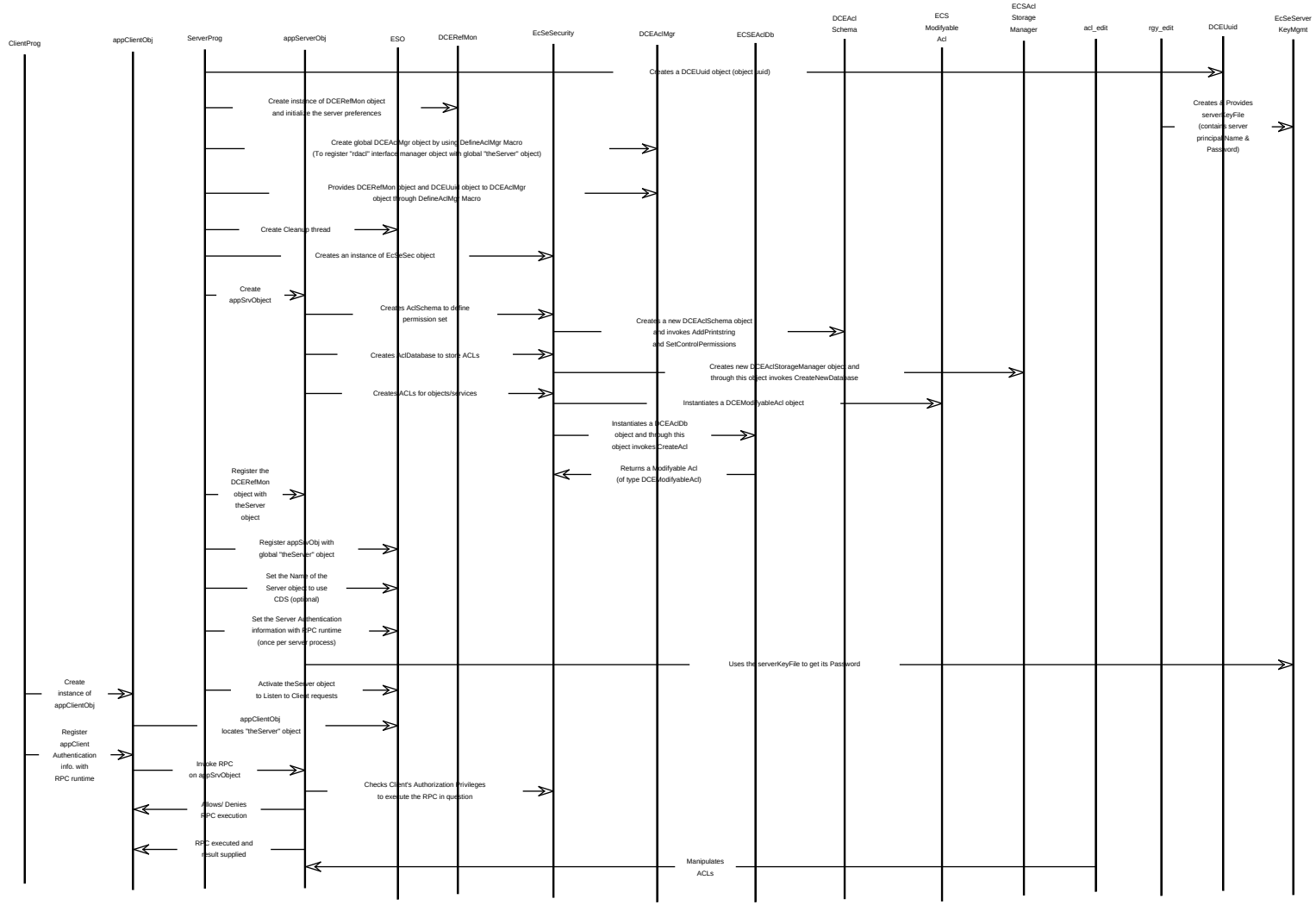

Figure 4.2.2.5-1. Security Event Trace

4.2.2.5.1.5 Participating Classes

CSS's EcSeSecurity and EcSeServerKeyMgmt classes.

COTS provided classes: DCEAclMgr, DCERefMon, ECSAcl, ECSAclDb, etc.

4.2.2.5.1.6 Pre-conditions

Segments implementing security need to run in the DCE environment.

4.2.2.5.1.7 Post-conditions

The database client and server authentication information are registered with their RPC runtime. A modifiable ACL with at least one person granted with the privilege to change ACL as desired, an ACL schema with default 7 permissions (read, write, execute, test, insert, delete, acl_control) and a provision to define 25 application specific permissions, an ACL database which has the latest updated ACL are available to the application server implementing security. The rdacl interface manager object is registered with theServer global object to provide the functionality to use MSS provided acl_edit utility to manipulate ACLs. A function to check client's authorization privileges to access resources is available.

4.2.2.5.1.8 Scenario Description

1. 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 appClObj;
```

c. Invoke the SetAuthInfo API

```
appClObj.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() 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.
```

2. 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 "EcSeServerKeyMgmt.H"
```

b. Establish Server Password

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

```
EcSeServerKeyMgmt  passwrd(serverPrincName, serverKeytabFilename);
```

c. Establish Server Login Context

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

```
DCEStdLoginContext  loginCntxt(&passwrd);
```

3. 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);
```

4. 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");
```

5. Initialization of AclManagement:

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.

6. Acl Management

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 EcSeSecurity 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 "EcSeSecurity.H"
```

b. Instantiate EcSeSecurity object

```
EcSeSecurity *ecsSecurityObj = new EcSeSecurity(char *databaseName);
```

c. Create acls, aclschema and acl database

Through this object can invoke the following functions:

```
ecsSecurityObj->CreateAclSchema();
DCEAclSchema *ecsSecurity-> GetSchema();
ECSAclDb *ecsSecurityObj->CreateDatabase(databaseName, *_Schema, char
                                                        *persistentDbName);
```

d. 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).

4.2.2.5.2 Scenario #2

4.2.2.5.2.1 Abstract

This scenario (see Figure 4.2.2.5-2) traces the interactions of the application developer with the DCE/OODCE provided "rgy_edit" utility, to create the server keytab file to store the server principal's (principal under who's identity the server is to run) password.

4.2.2.5.2.2 Interfaces

COTS provided rgy_edit application.

4.2.2.5.2.3 Stimulus

Application developer (dce authenticated user) invoke the rgy_edit function to create the server keytab file.

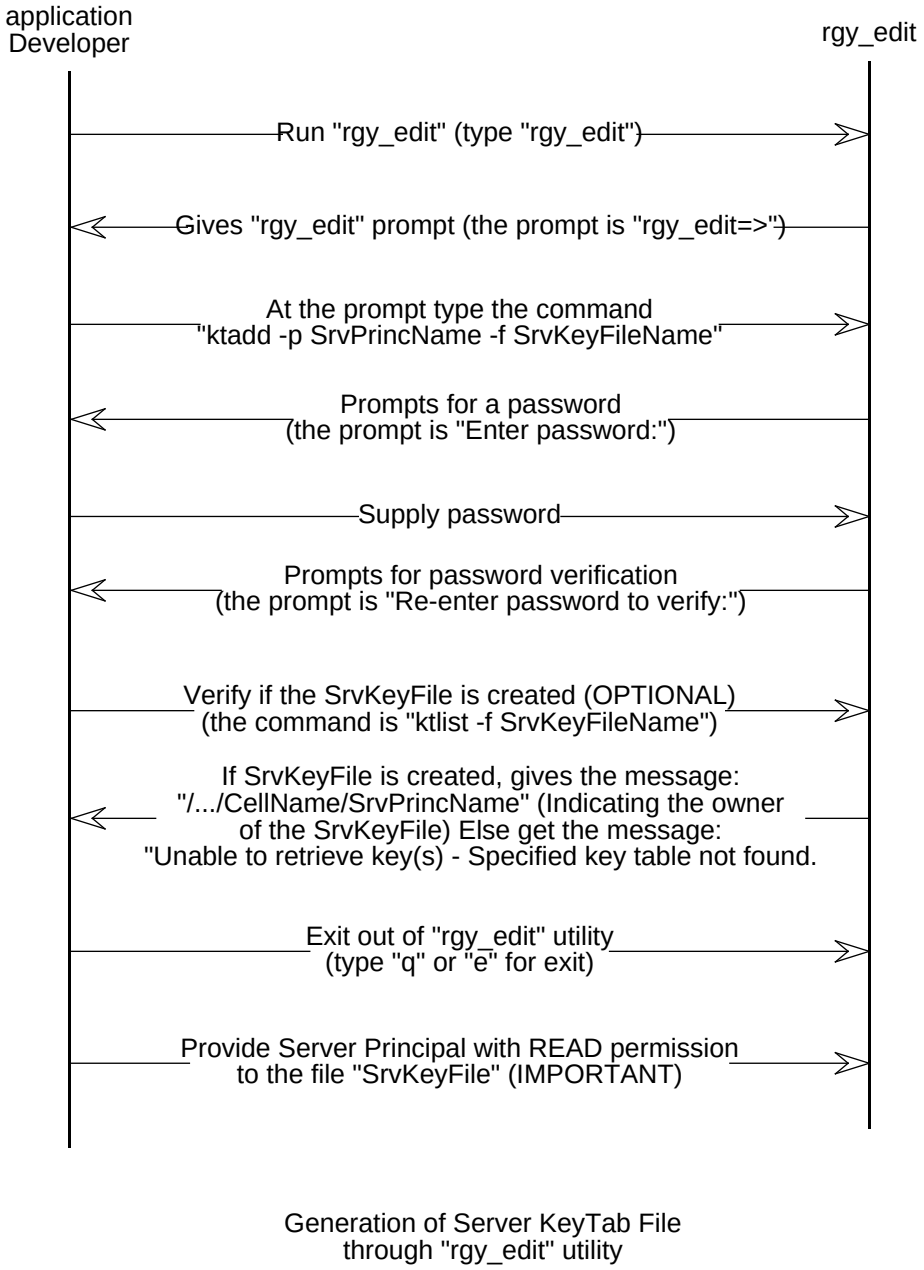

Figure 4.2.2.5-2. Security Event Trace

4.2.2.5.2.4 Desired Response

The server keytabfile should be created and available for use by the database server to get its password to establish its identity during login context.

4.2.2.5.2.5 Participating Classes

rgy_edit

4.2.2.5.2.6 Pre-conditions

The application developer should be a DCE authenticated user in order to create the keytab file through "rgy_edit".

4.2.2.5.2.7 Post-conditions

Then serverKeyTabFile is available for use by the database server to establish its identity during login context.

4.2.2.5.2.8 Scenario 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, please refer to DCE Administration Guide.

4.2.2.5.3 Scenario #3

4.2.2.5.3.1 Abstract

This scenario (see Figure 4.2.2.5-3) traces the interactions of M&O staff modifying the contents of an ACL associated with an access operation of a database server through an MSS-provided interface.

4.2.2.5.3.2 Interfaces

MSS's acl_edit application.

4.2.2.5.3.3 Stimulus

M&O staff invoke the acl_edit function to edit the ACL associated with given resource.

4.2.2.5.3.4 Desired Response

The server should use the modified ACL in any future authorization checks performed in deciding whether a given client can access the resource associated with the ACL.

4.2.2.5.3.5 Participating Classes

acl_edit

4.2.2.5.3.6 Pre-conditions

The database server is running and has implemented the rdacl calls associated with the database server.

4.2.2.5.3.7 Post-conditions

The newly modified ACL will be attached to the resource.

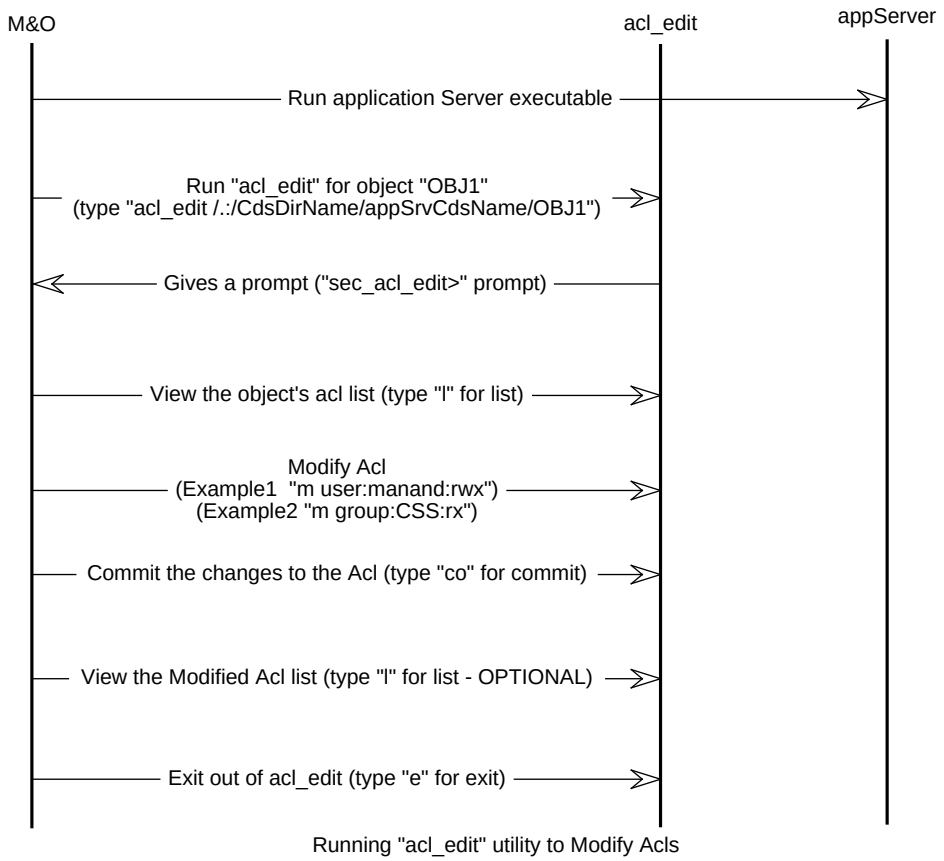

Figure 4.2.2.5-3. Security Event Trace

4.2.2.5.3.8 Scenario 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.

4.2.2.6 Implementation

The OODCE COTs product will be used for the security functionalities defined in this section. However, CSS will provide the following custom software:

- EcSeSecurity

  Provides a layer encapsulating the underlying security classes such as ECSAcl, ECSAclDb, ECSModifyableAcl, ECSAclStorageManager, DCEAclMgr, DCEAclSchema and DCESecId.

  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 an ACL is updated it is written into persistent storage file and whenever the 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.

- ECSPassword

  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.

4.2.2.7 Service/CSCI Management and Operation

4.2.2.7.1 System Management and Strategy

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.

4.2.2.7.2 Operator Interface

- 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.

For more information regarding these utilities, please refer to the DCE administration guide.

4.2.2.7.3 Reports

Not applicable.

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.

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.

Table 4.2.3-1. Message Passing Communication Types Defined

Type	Description
Synchronous	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. 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.
Non Guaranteed Asynchronous	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.
Guaranteed Asynchronous	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. 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.
Deferred Synchronous	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. As in guaranteed asynchronous message passing, the argument types are restricted to strings only.

4.2.3.1 Message Passing Implementation One

4.2.3.1.1 Overview

Implementation one 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.

4.2.3.1.3 Object Model

See Figure 4.2.3.1.3-1.

4.2.3.1.3.1 EcMpMsgCb Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class will handle two types of callbacks: 1. For ordinary receive messages:If an ordinary message is received, then handle Msg 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 send a message. Each sender session will have a logical name that is needed to contact the receiver. A list of sending sessions is maintained in a given application. A sender session list contains a callback object which provides virtual funtions to be called when a send is complete. This is done at the sender side. A callback object is created and will implement the acknowledgment. The virtual funtion 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).

Attributes:

None

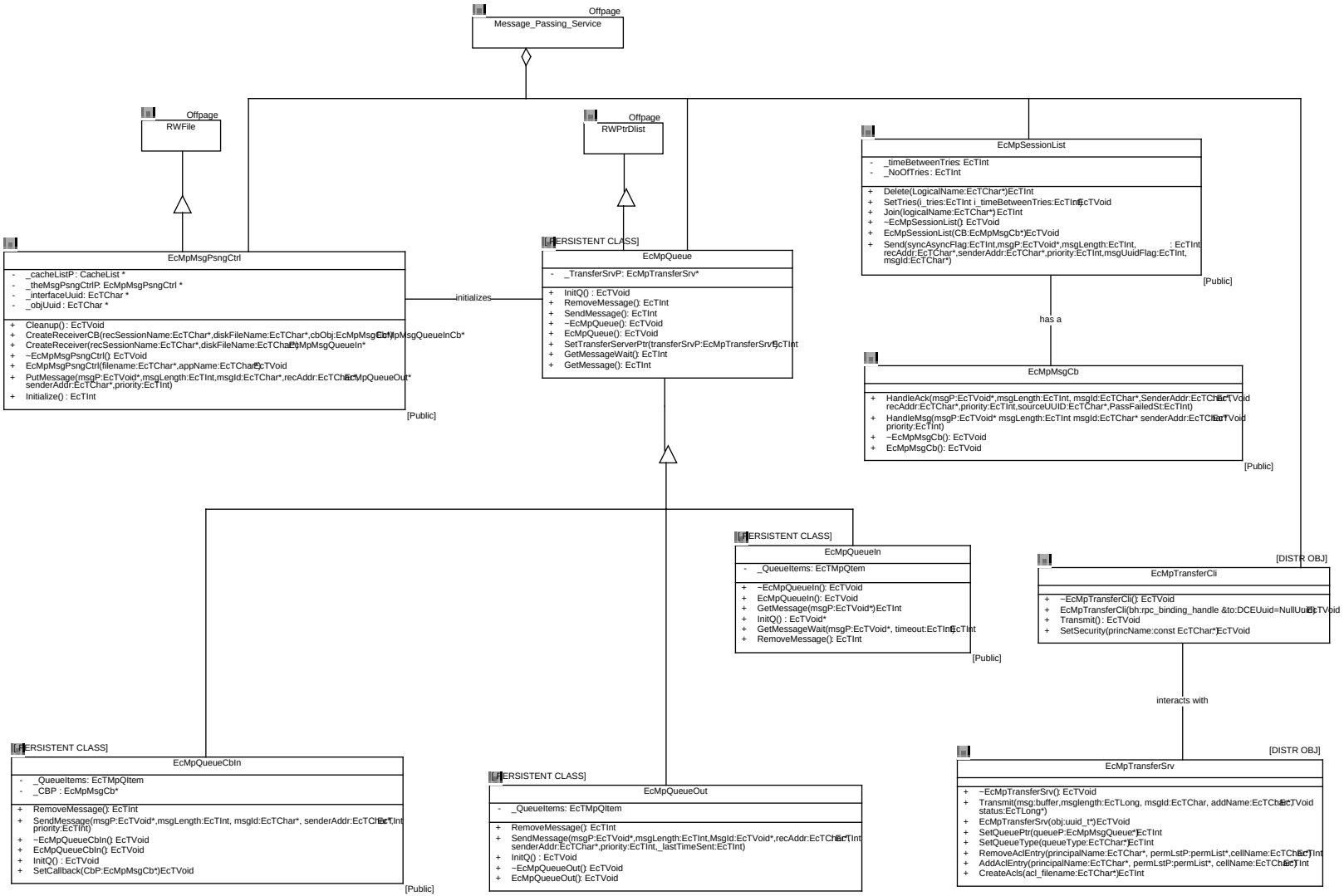

Figure 4.2.3.1.3 -1 RelA_MsgPassing_ObjectModel_1Object Model Diagram

Operations:

EcMpMsgCb - Constructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

HandleAck - Callback for 'send with acknowledgment', an asynchronous type of send.
Arguments:msgP:EcTVoid*,msgLength:EcTInt,
msgId:EcTChar*,SenderAddr:EcTChar*,
recAddr:EcTChar*,priority:EcTInt,sourceUUID:EcTChar*,PassFailedSt:EcTInt
Return Type:EcTVoid
Privilege:Public

HandleMsg - Callback for ordinary receive.
Arguments:msgP:EcTVoid* msgLength:EcTInt msgId:EcTChar* senderAddr:EcTChar* priority:EcTInt
Return Type:EcTVoid
Privilege:Public

~EcMpMsgCb - Destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpMsgCb class has associations with the following classes:
Class: EcMpSessionList hasa

4.2.3.1.3.2 EcMpMsgPsngCtrl Class

Parent Class:RWFile
Public:Yes
Distributed Object:No
Purpose and Description:
This class 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 (optional).The file is used for persistence. When a message comes in, it is stored in the queue associated with the object and a copy of it is stored in the file associated with this object. Internally, this object 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.

Attributes:

_cacheListP - This attribute is a two dimensional array of a hash table containg a list of pairs, an application name and the corresponding proxy object it is pointing to. The pair gets removed from the list when we find out that the server is not longer listening.
Data Type:CacheList *
Privilege:Private
Default Value:

_interfaceUuid - This attribute represents an interface uuid.
Data Type:EcTChar *
Privilege:Private
Default Value:

_objUuid - This attribute represents an object uuid.
Data Type:EcTChar *
Privilege:Private
Default Value:

_theMsgPsngCtrlP - This attribute is a pointer to the EcMpMsgPsngCtrl object, a global one.
Data Type:EcMpMsgPsngCtrl *
Privilege:Private
Default Value:

Operations:

Cleanup - This operation deallocates any memory used and destroys/shutdowns the threads.
Arguments:
Return Type:EcTVoid
Privilege:Public

CreateReceiver - A receiver is created by means of the CreateReceiver operation. Each receiver session is associated with a unique name (so other applications can send messages to this receiver), and a unique file. The file name is used for persistence. When a message comes in, it is stored into the queue associated with the object and a copy of it is stored on the file associated with this object. CreateReceiver will create/initialize the ordinary receiver queue, and open the file for persistence purposes. It will also create an instance of the transfer server object, and will attach a pointer to the proper queue.
Arguments:recSessionName:EcTChar*,diskFileName:EcTChar*
Return Type:EcMpMsgQueueIn*
Privilege:Public

- A special receiver can be created by means of this operation, it calls the user-defined call back function. A thread, just one, will be awakened every time a message arrives into the special reciver queue created by the call 'CreateReceiverCB'and a call back (a virtual function) will be executed. Then, it will check if there is any other message in the queue and if so, execute the call back again or else shutdown the thread until a message arrives in the queue. CreateReceiverCb will create/initialize the callback receiver queue and open its respective file for persistence purposes. It will also create an instance of the transfer server object, and it will attach a pointer to the proper queue.
Arguments:recSessionName:EcTChar*,diskFileName:EcTChar*,cbObj:EcMpMsgCb*
Return Type:EcMpMsgQueueInCb*
Privilege:Public

EcMpMsgPsngCtrl - This operation is the constructor. It takes two parameters: a filename and an application name. The file name will be used to name: . the five outgoing queues, one for each priority type . the five outgoing disk storage files - persistence . the incoming queue . the incoming disk storage file - persistence The application name will be a full CDS entry path name. It will represent the relative name in the CDS where the server will store its binding information. The caller's application store its binding in this application name. The CDS directory path should be created by default in the developer's cell with public write permissions. The caller needs to be authenticated in order to write the bindings into the CDS. At construction time the following actions take place: . Initialize the Outgoing/ Incoming queues. . Open the in/out disk files with read/write permissions. . Set the thread scheduling attributes for outogoing queues . Using appl. name,checks whether a thread is listening.
Arguments:filename:EcTChar*,appName:EcTChar*
Return Type:EcTVoid
Privilege:Public

Initialize - This operation starts listening if it is not already.
Arguments:
Return Type:EcTInt
Privilege:Public

PutMessage - This operation calls the EcMpQueueOut method Send to send a message.
Arguments:msgP:EcTVoid*,msgLength:EcTInt,msgId:EcTChar*,recAddr:EcTChar*, senderAddr:EcTChar*,priority:EcTInt
Return Type:EcMpQueueOut*
Privilege:Public

~EcMpMsgPsngCtrl - This operation is the default destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpMsgPsngCtrl class has associations with the following classes:
Class: EcMpQueue initializes
Message_Passing_Service (Aggregation)

4.2.3.1.3.3 EcMpQueue Class

Parent Class:RWPtrDlist
Public:No
Distributed Object:No
Persistent Class:True
Purpose and Description:
This class will be the parent class for the following: EcMpMsgQueueIn EcMpMsgQueueCbIn EcMpMsgQueueOut

Attributes:

_TransferSrvP - This attribute represents a pointer to the EcMpTransferSrv object.
Data Type:EcMpTransferSrv*
Privilege:Private
Default Value:

Operations:

EcMpQueue - This operation is the constructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

GetMessage - This operation performs a single read operation and returns a message from the Incoming queue or a return status. This call will visit the next node in the queue list. If an attempt was made to read beyond the tail, it will return CsCRemoveQueueFailed. If the queue is empty, it will return CsCQueueIsEmpty. If the read operation was successful, it will return CsCSuccess.
Arguments:
Return Type:EcTInt
Privilege:Public

GetMessageWait - This operation 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. It actually removes the head node from the queue. Returns either CsCSuccess, CsCRemoveQueueFailed, or CsCTimeout.
Arguments:

Return Type:EcTInt
Privilege:Public

InitQ - This operation initializes the queue. Calls RWTPtrDlist<List> to construct an empty list. It does also any other initialization steps that apply to the class EcMpMsgQueue.
Arguments:
Return Type:EcTVoid
Privilege:Public

RemoveMessage - This operation removes all the flagged items. This function checks for those messages with a deletion flag on, and calls the RW remove call to delete the message from the queue. This function will be called periodically.
Arguments:
Return Type:EcTInt
Privilege:Public

SendMessage - This operation sends a message.
Arguments:
Return Type:EcTInt
Privilege:Public

SetTransferServerPtr - This operation sets a pointer to the transfer server class.
Arguments:transferSrvP:EcMpTransferSrv*
Return Type:EcTInt
Privilege:Public

~EcMpQueue - This operation is the default destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpQueue class has associations with the following classes:
Class: EcMpMsgPsngCtrl initializes
Message_Passing_Service (Aggregation)

4.2.3.1.3.4 EcMpQueueCbIn Class

Parent Class:EcMpQueue
Public:Yes
Distributed Object:No
Persistent Class:True
Purpose and Description:
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.

Attributes:

_CBP - This attribute represents a callback pointer.
Data Type:EcMpMsgCb*
Privilege:Private
Default Value:

_QueueItems - This attribute represents a queue item type.
Data Type:EcTMpQItem
Privilege:Private
Default Value:

Operations:

EcMpQueueCbIn - This operation is the constructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

InitQ - This operation initializes the queue. Calls RWTPtrDlist<inCbList> to construct an empty list. It does also any other initialization steps that apply to the class EcMpMsgQueueCbIn.
Arguments:
Return Type:EcTVoid
Privilege:Public

RemoveMessage - This operation removes all the flagged items. This function checks for those messages with a deletion flag on, and calls the RW remove call to delete the message from the queue. This function will be called periodically.
Arguments:
Return Type:EcTInt
Privilege:Public

SendMessage - Once a message arrives in the queue, this operation will wake up a thread (there will be only one thread) and it will invoke the virtual callback function 'HandleMsg' to let the receiver know that a message just arrived in the queue. Next, it will search to check if another message arrived in the queue, and if so it will invoke another callback. If there are no messages in the queue, the thread will be shutdown until a message arrives in which case it will be awakened again.
Arguments:msgP:EcTVoid*,msgLength:EcTInt, msgId:EcTChar*, senderAddr:EcTChar*, priority:EcTInt

Return Type:EcTInt
Privilege:Public

SetCallback - This operation sets a pointer to the callback object.
Arguments:CbP:EcMpMsgCb*
Return Type:EcTVoid
Privilege:Public

~EcMpQueueCbIn - This operation is the default destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpQueueCbIn class has associations with the following classes:
None

4.2.3.1.3.5 EcMpQueueIn Class

Parent Class:EcMpQueue
Public:Yes
Distributed Object:No
Persistent Class:True
Purpose and Description:
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.

Attributes:

_QueueItems - This attribute represents a queue item type.
Data Type:EcTMpQtem
Privilege:Private
Default Value:

Operations:

EcMpQueueIn - Default constructor.
Arguments:

Return Type:EcTVoid
Privilege:Public

GetMessage - This operation performs a single read operation and returns a message from the incoming queue or a return status. This call will visit the next node in the queue list. If an attempt was made to read beyond the tail, it will return CsCRemoveQueueFailed. If the queue is empty, it will return CsCQueueIsEmpty. If the read operation was successful, it will return CsCSuccess.
Arguments:msgP:EcTVoid*
Return Type:EcTInt
Privilege:Public

GetMessageWait - This operation 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. It actually removes the head node from the queue and returns either CsCSuccess, CsCRemoveQueueFailed, or CsCTimeout.
Arguments:msgP:EcTVoid*, timeout:EcTInt
Return Type:EcTInt
Privilege:Public

InitQ - This operation initializes the queue. It calls RWTPtrDlist<inList> to construct an empty list. It does also any other initialization steps that apply to the class EcMpMsgQueueIn.
Arguments:
Return Type:EcTVoid*
Privilege:Public

RemoveMessage - This operation removes all the flagged items. This function checks for those messages with a deletion flag on, and calls the RW remove call to delete the message from the queue. This function will be called periodically.
Arguments:
Return Type:EcTInt
Privilege:Public

~EcMpQueueIn - Default destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpQueueIn class has associations with the following classes:
None

4.2.3.1.3.6 EcMpQueueOut Class

Parent Class:EcMpQueue
Public:No
Distributed Object:No
Persistent Class:True
Purpose and Description:
In case of asynchronous calls, the message is put in a queue, which will be sent later by an internal thread. Controls returns right away. 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' + ' _timeBetweenRetries' was reached or until '_noOfTries' expired. If the message failed to be sent, the message will be returned (the callback 'handleAck' will be invoked). The EcMpMsgQueueOut class defines a double link list queue. It inherits from the Rogue Wave library file, RWTPtrDlist.

Attributes:

_QueueItems - This attribute represents a queue item type.
Data Type:EcTMpQItem
Privilege:Private
Default Value:

Operations:

EcMpQueueOut - This operation is the constructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

InitQ - This operation initializes the queue. Calls RWTPtrDlist<outList> to construct an empty list. It does also any other initialization steps that apply to the class EcMpMsgQueueOut.
Arguments:
Return Type:EcTVoid
Privilege:Public

RemoveMessage - This operation removes all the flagged items. This function checks for those messages with a deletion flag on, and calls the RW remove call to delete the message

from the queue. This function will be called periodically.
Arguments:
Return Type:EcTInt
Privilege:Public

SendMessage - Performs a send operation (used only for asynchronous calls). It does not remove the item from the queue, it uses a pointer to the data. If the send call fails, 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 and until the '_noOfTries' expired. At that point, the caller will get a notification informing that the message failed to be sent. If the message was sent successfully, it will be flagged, so it can be deleted later on.
Arguments:msgP:EcTVoid*,msgLength:EcTInt,MsgId:EcTVoid*,recAddr:EcTChar*, senderAddr:EcTChar*,priority:EcTInt,_lastTimeSent:EcTInt
Return Type:EcTInt
Privilege:Public

~EcMpQueueOut - This operation is the default destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpQueueOut class has associations with the following classes:
None

4.2.3.1.3.7 EcMpSessionList Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This is a container class whose element type is a logical name and will inherit from the RWTPtrSlist class. 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.

Attributes:

_NoOfTries - Number of tries in case of communication errors.
Data Type:EcTInt
Privilege:Private
Default Value:

_timeBetweenTries - Number of seconds between tries.
Data Type:EcTInt
Privilege:Private
Default Value:

Operations:

Delete - Deletes a receiver, a logical name from the list.
Arguments:LogicalName:EcTChar*
Return Type:EcTInt
Privilege:Public

EcMpSessionList - Constructor.
Arguments:CB:EcMpMsgCb*
Return Type:EcTVoid
Privilege:Public

Join - Used to insert a logical name into the list.
Arguments:logicalName:EcTChar*
Return Type:EcTInt
Privilege:Public

Send - Sends a message to a particular session. The message can be sent expecting an acknowledgment or not expecting an acknowledgment. Internally if the flag is synchronous, the remote function, transmit, will be called and the message sent at once. Control will return when the Send operation completed. If the flag is asynchronous, the message will be placed in the outgoing queue and it will be sent from there.
Arguments:syncAsyncFlag:EcTInt,msgP:EcTVoid*,msgLength:EcTInt,
recAddr:EcTChar*,senderAddr:EcTChar*,priority:EcTInt,msgUuidFlag:EcTInt,
msgId:EcTChar*
Return Type:EcTInt
Privilege:Public

SetTries - Set the number of tries if a message fails to be sent and needs to be retried, and the time between the tries.
Arguments:i_tries:EcTInt i_timeBetweenTries:EcTInt
Return Type:EcTVoid
Privilege:Public

~EcMpSessionList - Destructor.
Arguments:
Return Type:EcTVoid

Privilege:Public

Associations:

The EcMpSessionList class has associations with the following classes:
Class: EcMpMsgCb hasa
Message_Passing_Service (Aggregation)

4.2.3.1.3.8 EcMpTransferCli Class

Parent Class:Not Applicable
Public:No
Distributed Object:Yes
Purpose and Description:
Class EcMpTransferCli is a surrogate object. This class is the 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 as you ask to to perform a transfer. This class allows you to transfer.

Attributes:

None

Operations:

EcMpTransferCli - Constructor.
Arguments:bh:rpc_binding_handle &to:DCEUuid=NullUuid
Return Type:EcTVoid
Privilege:Public

SetSecurity - Set security levels. The default is to ask for secret key authentication, packet integrity level protection, and the name authorization service. If a NULL pointer is given or a NULL string, then security is not set.
Arguments:princName:const EcTChar*
Return Type:EcTVoid
Privilege:Public

Transmit - Call the transmit function.
Arguments:
Return Type:EcTVoid
Privilege:Public

~EcMpTransferCli - Default destructor.
Arguments:

Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpTransferCli class has associations with the following classes:
Class: EcMpTransferSrv interactswith
Message_Passing_Service (Aggregation)

4.2.3.1.3.9 EcMpTransferSrv Class

Parent Class:Not Applicable
Public:No
Distributed Object:Yes
Purpose and Description:
This class is the EcMpTransferSrv manager object. It responds to client's requests to transfer data which results in the enqueuing of the data.

Attributes:

None

Operations:

AddAclEntry - Add entries to the ACL database.
Arguments:principalName:EcTChar*, permLstP:permList*, cellName:EcTChar*
Return Type:EcTInt
Privilege:Public

CreateAcls - Set ACL permissions.
Arguments:acl_filename:EcTChar*
Return Type:EcTInt
Privilege:Public

EcMpTransferSrv - Constructor taking an object uuid.
Arguments:obj:uuid_t*
Return Type:EcTVoid
Privilege:Public

RemoveAclEntry - Remove entries from the ACL database
Arguments:principalName:EcTChar*, permLstP:permList*,cellName:EcTChar*
Return Type:EcTInt
Privilege:Public

SetQueuePtr - Set queue pointer.
Arguments:queueP:EcMpMsgQueue*
Return Type:EcTInt
Privilege:Public

SetQueueType - Set queue type.
Arguments:queueType:EcTChar*
Return Type:EcTInt
Privilege:Public

Transmit - This is the read-side RPC manager function; this is the function that actually implements the remote procedure defined in the .idl file. The server stub (called by the RPC runtime) calls this function when an RPC request comes in for this interface. The manager function takes the arguments defined in the .idl file, performs its function and returns results as defined in the .idl file. The .idl file specifies use of explicit binding the manager must take a binding handle as its first argument. Note: the code in this manager function must be (and is) reentrant as it may be running simultaneously in multiple server threads. Transmit will enqueue the message in the proper queue.
Arguments:msg:buffer,msglength:EcTLong, msgId:EcTChar, addName:EcTChar*, status:EcTLong*
Return Type:EcTVoid
Privilege:Public

~EcMpTransferSrv - Destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcMpTransferSrv class has associations with the following classes:
Class: EcMpTransferCli interactswith

4.2.3.1.3.10 Message_Passing_Service Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
This implementation will provide for asynchronous and synchronous message passing with store and forward, and recovery - persistence. The concept of groups is included into this API. A group is a synbolic name and it represnets a nu ber of receivers. Each receiver will be identified by a logical name associated to a UUID. This API will handle messages consisting of only byte streams.

Attributes:

None

Operations:

None

Associations:

The Message_Passing_Service class has associations with the following classes:
None

4.2.3.1.3.11 RWFile Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Rogue Wave class which encapsulates binary file operations using the Standard C stream library.

Attributes:

None

Operations:

None

Associations:

The RWFile class has associations with the following classes:
None

4.2.3.1.3.12 RWPtrDlist Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Rogue Wave class which mantains a collection of pointers to type T, implemented as a doubly linked list. This is a pointer based list: pointers to objects are copied in and out of

the links that make up the list.

Attributes:

None

Operations:

None

Associations:

The RWPtrDlist class has associations with the following classes:
None

4.2.3.1.4 Functional Model

In order to achieve asynchronous and synchronous message passing, 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 as depicted in the Figure 4.2.3.1.3-1.

In asynchronous message passing, when the client sends the data to a set of receiving processes, a pointer to the data is kept in intermediate message queues. 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.

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.

A distributed object (DO) is defined with the functionality to transfer messages. After selecting a message from a queue, a thread creates a DO 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 DO 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 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.

When a message is to be sent, a pointer to it is placed in to an outgoing queue. 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.

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.

Some constraints are: 1) Each queue must have a maximum number of messages, a user-defined number; 2) The messages must be of defined maximum length; 3) Logical names ought to be unique, internally CDS/GDS will be used to store location information associated the receiver (logical name-UUID pair); 4) The message or buffer should not be changed while the sending is in progress; otherwise, we need to copy the message; 5) 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.

4.2.3.1.5 Dynamic Model Scenarios

4.2.3.1.5.1 Scenario #1

4.2.3.1.5.1.1 Abstract

This scenario will demonstrate how a process can be a sender of asynchronous messages and a receiver as well.

4.2.3.1.5.1.2 Interfaces

OODCE provided classes.

4.2.3.1.5.1.3 Stimulus

The process wants to send a message to another process and be able to receive as well.

4.2.3.1.5.1.4 Desired Response

A message is sent and a message is received.

4.2.3.1.5.1.5 Participating Classes

EcMpMsgCtrl, EcMpMsgCb, EcMpSessionList, EcMpQueue, EcMpQueueIn, EcMpQueueOut, EcMpTransferSrv, EcMpTransferCli.

4.2.3.1.5.1.6 Pre-conditions

The user must determine the receiver identity and under which identity the process wishes to receive messages.

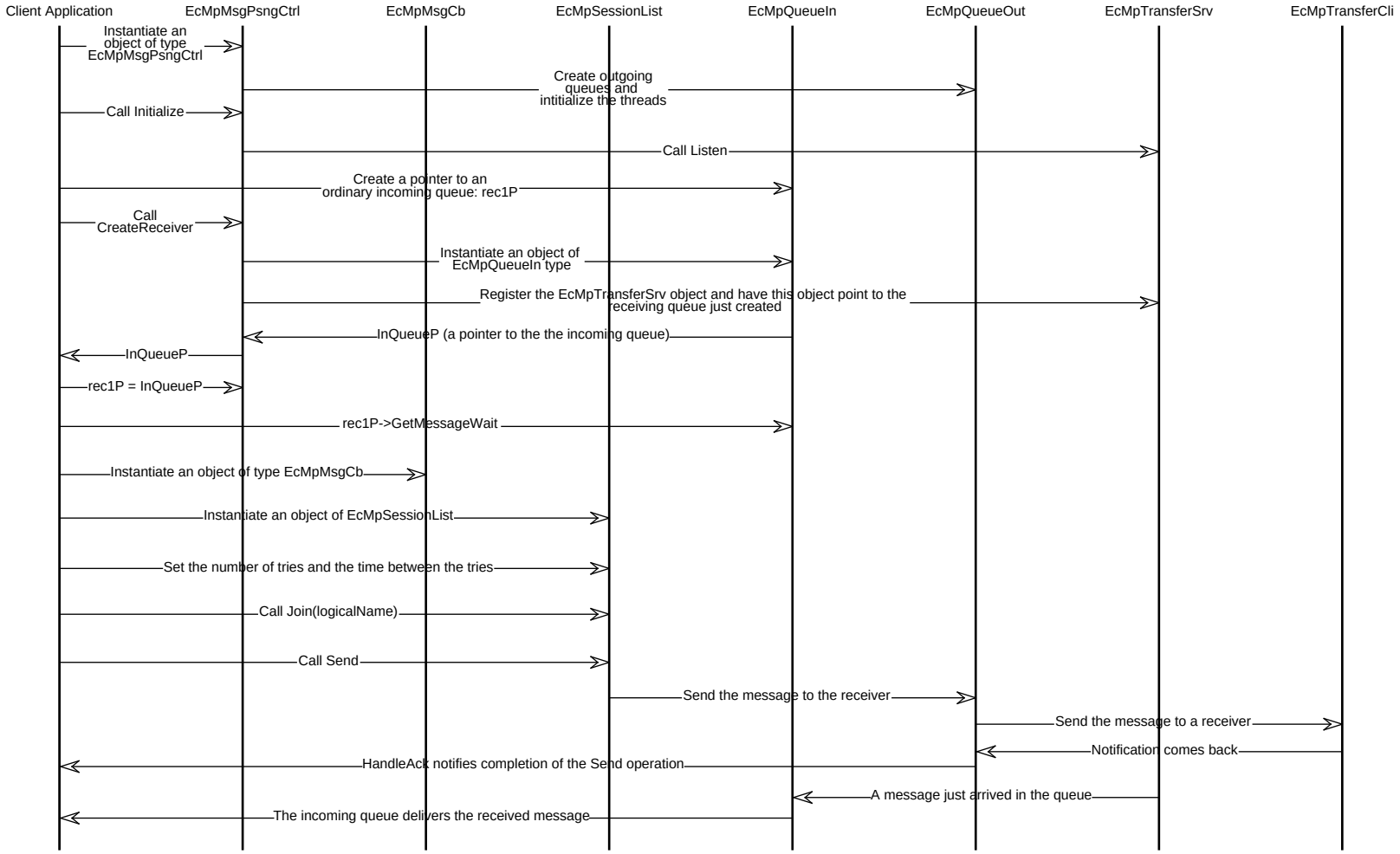

Figure 4.2.3.1.5-1 Message Passing Scenario One Event Trace

Table 4.2.3.1.5-1 Message Passing Object Responsibility Matrix (1 of 2)

Class Name	Description
EcMpMsgPsngCtrl	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 an optional unique file. The file is used for persistence. 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. 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 initializing call which periodically get messages from the outgoing queue and send them.
EcMpMsgCb	This class will handle two types of callbacks: 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. 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. 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).
EcMpSessionList	This is a container class whose element type is a logical name and will inherit from the RWTPtrSlist class. 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.
EcMpQueue	This class will be the parent class for the following: . EcMpMsgQueueIn . EcMpMsgQueueCbIn . EcMpMsgQueueOut EcMpMsgQueue inherits from the Rogue Wave library file RWPtrDlist.
EcMpQueueCbIn	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.

Table 4.2.3.1.5-1 Message Passing Object Responsibility Matrix (2 of 2)

Class Name	Description
EcMpQueueIn	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.
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 EcMpMsgQueueOut 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 enqueuing 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.

4.2.3.1.5.1.7 Post-conditions

The message is delivered, and the notification is passed back to the sender.

A message is received from another process and it is retrieved.

4.2.3.1.5.1.8 Scenario Description

a. <u>Set up an ordinary receiver session</u>

Instantiate an object of the EcMpMsgPsngCtrl 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.

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 to 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 to the sender that can be used by the callbacks or to get a reply back.

4.2.3.1.5.1.9 Event Trace

See Figure 4.2.3.1.5-2.

4.2.3.1.5.2 Scenario #2

4.2.3.1.5.2.1 Abstract

This scenario will demonstrate the usage of a callback to receive a message. The receiver will be notified everytime a message is received.

4.2.3.1.5.2.2 Interfaces

OODCE provided classes.

4.2.3.1.5.2.3 Stimulus

The arrival of a message.

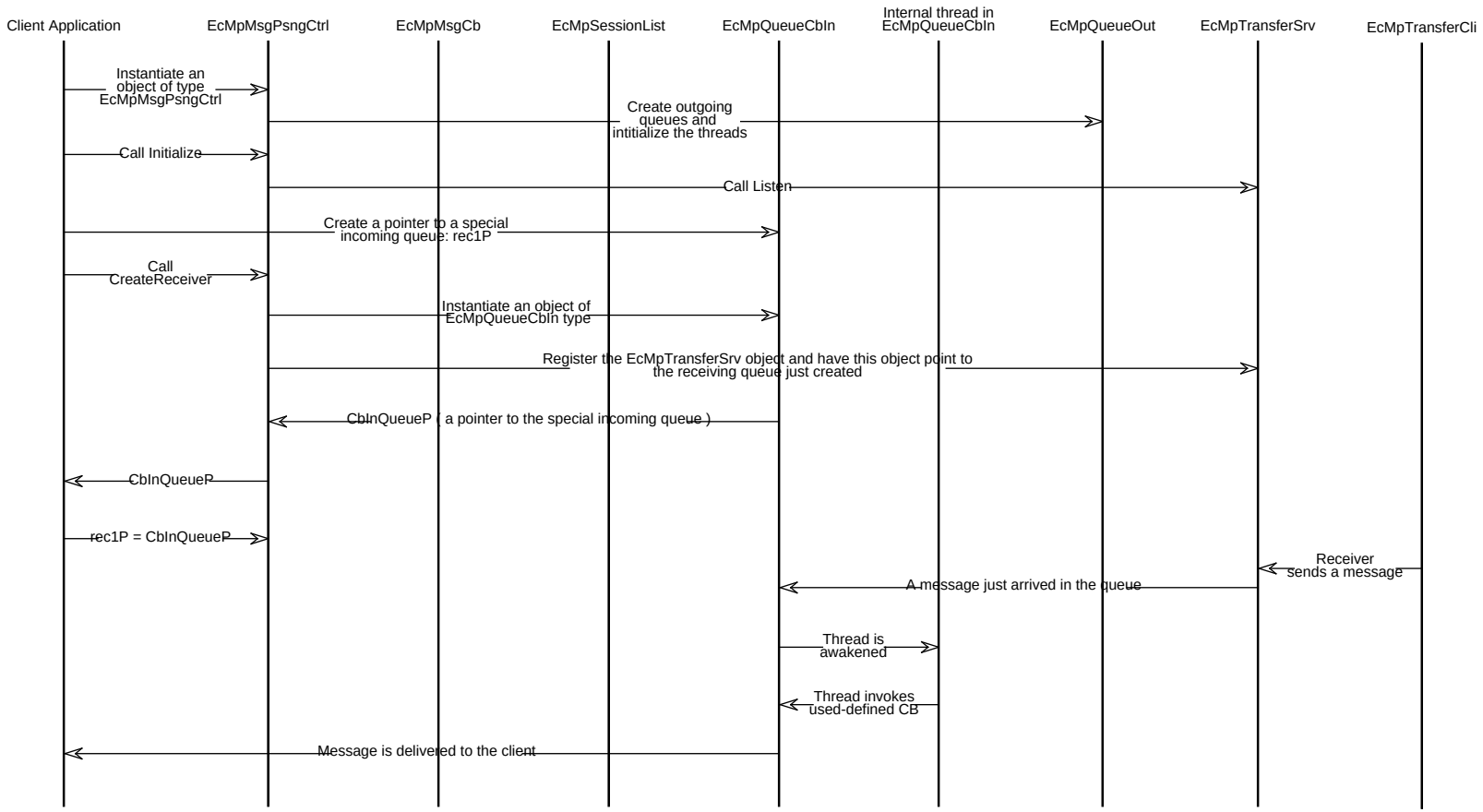

Figure 4.2.3.1.5-2 Message Passing Scenario Two Event Trace

4.2.3.1.5.2.4 Desired Response

A thread is awakened when a message is received in the queue. A callback is invoked at that point.

4.2.3.1.5.2.5 Participating Classes

EcMpMsgCtrl, EcMpMsgCb, EcMpSessionList, EcMpQueue, EcMpQueueCbIn, EcMpQueue-Out, EcMpTransferSrv, EcMpTransferCli.

4.2.3.1.5.2.6 Pre-conditions

The user must determine the receiver identity and under which identity the process wishes to receive messages.

4.2.3.1.5.2.7 Post-conditions

A callback notification is received informing the user a message arrived.

4.2.3.1.5.2.8 Scenario Description

Setup a callback receiver session

Instantiate an object of the EcMpMsgPsngCtrl 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.

4.2.3.1.5.2.9 Event Trace

4.2.3.1.6 Implementation

This service will utilize COTS such as OODCE and RogueWave.

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

4.2.3.1.7 Service/CSCI Management and Operation

4.2.3.1.7.1 System Management and Strategy

Not Applicable.

4.2.3.1.7.2 Operator Interface

Not Applicable.

4.2.3.1.7.3 Reports

None.

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 implementation two would need to create a thread to listen for a reply from the server. Reply sensing will be COTS with CORBA. In addition, recovery is not supported with implementation two.

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.

4.2.3.2.3 Object Model

```
EcDcDSyncCom
-------------------------------------------------------------------------
- _policy : EcEDcThreadPolicy = EcDDcFg
- _priority : EcEDcThreadPriority = EcDDcPri_min
- _results : EcTPtr = NULL
- _addr : EcTPtr = NULL
- _data : EcTPtr = NULL
- _call_in_progress : EcTInt = 0
- _done : EcTInt = 0
- _timeBetweenTries : EcTInt = 0
- _noOfTries : EcTInt = 0
-------------------------------------------------------------------------
+ SetCallInProgressFlag(a_call_in_progress:EcTInt) : EcUtStatus
+ SetDoneFlag(a_done:EcTInt) : EcUtStatus
+ EvalThread(a_param:DCEPthreadParam) : DCEPthreadResult
+ Done(ao_status:EcUtStatus*) : EcTInt
+ Send(a_return_uuid_flag:EcTInt,ao_status:EcUtStatus*) : EcTChar*
+ PostInvoke() : EcTInt
+ Invoke() : EcTInt
+ PreInvoke() : EcTInt
+ CallInProgress(ao_status:EcUtStatus*) : EcTInt
+ GetResults(ao_status:EcUtStatus*) : EcTPtr
+ SetResults(a_resultP:EcTPtr) : EcUtStatus
+ GetTimeBetweenTries(ao_status:EcUtStatus*) : EcTInt
+ SetTimeBetweenTries(a_timeBetweenTries:EcTInt) : EcUtStatus
+ GetNoOfTries(ao_status:EcUtStatus*) : EcTInt
+ SetNoOfTries(a_noOfTries:EcTInt) : EcUtStatus
+ GetPolicy(ao_status:EcUtStatus*) : EcEDcThreadPolicy
+ SetPolicy(a_policy:EcEDcThreadPolicy) : EcUtStatus
+ GetPriority(ao_status:EcUtStatus*) : EcEDcThreadPriority
+ SetPriority(a_priority:EcEDcThreadPriority) : EcUtStatus
+ GetAddr(ao_status:EcUtStatus*) : EcTPtr
+ SetAddr(a_addrP:EcTPtr) : EcUtStatus
+ GetData(ao_status:EcUtStatus*) : EcTPtr
+ SetData(a_dataP:EcTPtr) : EcUtStatus
+ ~EcDcDSyncCom() : EcTVoid
+ EcDcDSyncCom() : EcTVoid
+ Reset() : EcUtStatus
-------------------------------------------------------------------------
                                                              [Public]
```

Figure 4.2.3.2.3-1 Message Passing Object Model

4.2.3.2.3.1 EcDcDSyncCom Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
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.

Attributes:

_addr - This attribute points to some address that the developer will use in the overridden Invoke member function. It is a void pointer and it could be a port number, an IP, a binding, an object reference, a CDS name, etc.
Data Type:EcTPtr
Privilege:Private
Default Value:NULL

_call_in_progress - This attribute holds a value that identifies whether a thread is currently executing or not. This flag is used to assure that only one call is processed at a time, and that we don't have multiple Send calls happening concurrenlty. Only one thread will be running at a time.
Data Type:EcTInt
Privilege:Private
Default Value:0

_data - This attribute points to some data that the developer will use in the overridden invoke member function. It is a void pointer.
Data Type:EcTPtr
Privilege:Private
Default Value:NULL

_done - This attribute holds a value that identifies whether a thread has terminated or not. It gets updated when the process finished execution, after PostInvoke. '0' means the thread did not finish, '1' means it did.
Data Type:EcTInt
Privilege:Private
Default Value:0

_noOfTries - This attribute defines the number of Send call re-tries if exceptions or errors occur during the communication.
Data Type:EcTInt
Privilege:Private
Default Value:0

_policy - This attribute represents the thread scheduling policy. It is an enum type. The policy types are: EcDDcFifo (first in/first out), EcDDcRr (Round Robin), EcDDcFg (Foreground), EcDDcBg (Background).
Data Type:EcEDcThreadPolicy
Privilege:Private
Default Value:EcDDcFg

_priority - This attribute represents the thread scheduling priority. It is an enum type. The priority types are: EcDDcPri_min, EcDDcPri_low, EcDDcPri_mid, EcDDcPri_hi, EcDDcPri_max.
Data Type:EcEDcThreadPriority
Privilege:Private
Default Value:EcDDcPri_min

_results - This attribute is used to store the results that were a product of the thread execution. It is a void pointer.
Data Type:EcTPtr
Privilege:Private
Default Value:NULL

_timeBetweenTries - This attribute defines how often Send call re-tries will occur in case of errors during the communication.
Data Type:EcTInt
Privilege:Private
Default Value:0

Operations:

CallInProgress - This operation returns to the caller the execution status of the thread, whether is currently executing (in progress), or not.
Arguments:ao_status:EcUtStatus*
Return Type:EcTInt
Privilege:Public

Done - This operation returns to the caller the thread termination status, an integer called '_done'. It is a flag and will be initialized to '0' in the constructor. When the thread finishes its execution, which is after PostInvoke, the '_done' flag is set to '1'. Once the results of the thread execution have been retrieved successfully, then Reset() can be used to clear the flag back to '0'.
Arguments:ao_status:EcUtStatus*
Return Type:EcTInt
Privilege:Public

EcDcDSyncCom - Constructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

EvalThread - This is a static member function, an internal one. It makes calls to PreInvoke, Invoke, and PostInvoke. This function will be executing on a separate thread. It takes as input parameter a DCEPthreadParam argument, and OODCE type which represents a void pointer.
Arguments:a_param:DCEPthreadParam
Return Type:DCEPthreadResult
Privilege:Public

GetAddr - This operation returns a pointer to the address.
Arguments:ao_status:EcUtStatus*
Return Type:EcTPtr
Privilege:Public

GetData - This operation returns a pointer to the data.
Arguments:ao_status:EcUtStatus*
Return Type:EcTPtr
Privilege:Public

GetNoOfTries - This operation returns the number of tries.
Arguments:ao_status:EcUtStatus*
Return Type:EcTInt
Privilege:Public

GetPolicy - This operation returns the thread scheduling policy attribute.
Arguments:ao_status:EcUtStatus*
Return Type:EcEDcThreadPolicy
Privilege:Public

GetPriority - This operation returns the thread priority attribute.
Arguments:ao_status:EcUtStatus*
Return Type:EcEDcThreadPriority
Privilege:Public

GetResults - This operation returns the method's results. This is applicable only for deferred synchronous. Once data was sent, a reply comes back with the results from the operation.
Arguments:ao_status:EcUtStatus*
Return Type:EcTPtr
Privilege:Public

GetTimeBetweenTries - This operation returns the time between tries.
Arguments:ao_status:EcUtStatus*
Return Type:EcTInt
Privilege:Public

Invoke - This operation is used to perform whatever operations the developer wishes. It is called automatically by the static function EvalThread, which is internal to CSS.
Arguments:
Return Type:EcTInt
Privilege:Public

PostInvoke - This operation is used to perform whatever operations the developer wishes. It is called automatically by the static function EvalThread, which is internal to CSS.
Arguments:
Return Type:EcTInt
Privilege:Public

PreInvoke - This operation is used to perform whatever operations the developer wishes. It is called automatically by the static function EvalThread, which is internal to CSS.
Arguments:
Return Type:EcTInt
Privilege:Public

Reset - It is called after the thread has terminated, and after the results from the thread execution (if any) were retrieved. This call will reset the '_done' flag to zero, the '_call_in_progress' flag to zero, and it will deallocate any memory assigned to '_results'.
Arguments:
Return Type:EcUtStatus
Privilege:Public

Send - When the Send operation is called, a thread gets created and its scheduling attributes are set. If the Send call fails, the call will be retried as many times as the developer defined in the '_noOfRetries' field. Once the thread began execution, the '_call_in_progress' flag is set to '1', the thread is terminated (but the object is not deleted), and the 'call_in_progress' flag is set to '0'. If the call was deferred synchronous, the method's results will be stored in '_results'.
Arguments:a_return_uuid_flag:EcTInt,ao_status:EcUtStatus*
Return Type:EcTChar*
Privilege:Public

SetAddr - This operation sets a pointer to some address that the developer wishes to use in the overridden Invoke member function. The address can be an IP, a port number, an object reference, a binding, a CDS name, etc.
Arguments:a_addrP:EcTPtr
Return Type:EcUtStatus

Privilege:Public

SetCallInProgressFlag - This operation sets the '_call_in_progress' flag. This flag is used to identify whether a thread is currently executing or not.
Arguments:a_call_in_progress:EcTInt
Return Type:EcUtStatus
Privilege:Public

SetData - This operation sets a pointer to some data that the developer wishes to use in the overridden Invoke member function.
Arguments:a_dataP:EcTPtr
Return Type:EcUtStatus
Privilege:Public

SetDoneFlag - This operation sets the '_done' flag. This flag is used to identify whether a thread has finished or not.
Arguments:a_done:EcTInt
Return Type:EcUtStatus
Privilege:Public

SetNoOfTries - This operation sets the number of Send retries in case of exceptions/ communications errors at the other end. The Send call will be retried as many times as specified in '_noOfTries'.
Arguments:a_noOfTries:EcTInt
Return Type:EcUtStatus
Privilege:Public

SetPolicy - This operation sets the thread scheduling policy attribute. The scheduling policies are: EcDDcFifo (first in/first out), EcDDcRr (Round Robin), EcDDcFg (Foreground), EcDDcBg (Background).
Arguments:a_policy:EcEDcThreadPolicy
Return Type:EcUtStatus
Privilege:Public

SetPriority - Threads are scheduled according to their scheduling priority and how the scheduling policy treats those priorities. This operation sets the thread priority attribute. The thread scheduling priorities are: EcDDcPri_min, EcDDcPri_low, EcDDcPri_mid, EcDDcPri_hi, EcDDcPri_max.
Arguments:a_priority:EcEDcThreadPriority
Return Type:EcUtStatus
Privilege:Public

SetResults - This operation sets the result from the Invoke call.
Arguments:a_resultP:EcTPtr
Return Type:EcUtStatus

Privilege:Public

SetTimeBetweenTries - This operation sets the time between each Send retry (in seconds). If the Send call failed because of communication errors, the Send call will be retried as often as specified in '_timeBetweenTries'.
Arguments:a_timeBetweenTries:EcTInt
Return Type:EcUtStatus
Privilege:Public

~EcDcDSyncCom - Destructor.
Arguments:
Return Type:EcTVoid
Privilege:Public

Associations:

The EcDcDSyncCom class has associations with the following classes:
None

4.2.3.2.4 Functional Model

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 priory set aside. The programmer can then either send some more data or terminate and delete the client object.

4.2.3.2.5 Dynamic Model Scenarios

4.2.3.2.5.1 Scenario #1

4.2.3.2.5.1.1 Abstract

This scenario will demonstrate how a process can be a sender of deferred synchronous messages and at the same time operate as a server (listen for incoming requests).

4.2.3.2.51.2 Interfaces

OODCE provided classes

4.2.3.2.51.3 Stimulus

The process wants to send a message to another process, and get a reply back. It also listens for incoming requests.

4.2.3.2.51.4 Desired Response

A message is sent and its result comes back.

4.2.3.2.51.5 Participating Classes

EcDcDSyncCom

4.2.3.2.51.6 Pre-conditions

The user must determine the receiver identity and under which identity the process listens.

4.2.3.2.51.7 Post-conditions

The message is delivered, and the notification is passed back. The result comes back after the send operation completed.

Table 4.2.3.2.4-1 Message Passing Object Responsibility Matrix

Class Name	Description
EcDcDSyncCom	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. 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. Provides the functionality to - 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.

4.2.3.2.5.8 Scenario Description

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.

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.

When calling 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.

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

4.2.3.2.5.1.9 Event Trace

See Figure 4.2.3.2.5-1.

4.2.3.2.6 Implementation

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.

4.2.3.2.7 Service/CSCI Management and Operation

4.2.3.2.7.1 System Management and Strategy

Not Applicable.

4.2.3.2.7.2 Operator Interface

Not Applicable.

4.2.3.2.7.3 Reports

None.

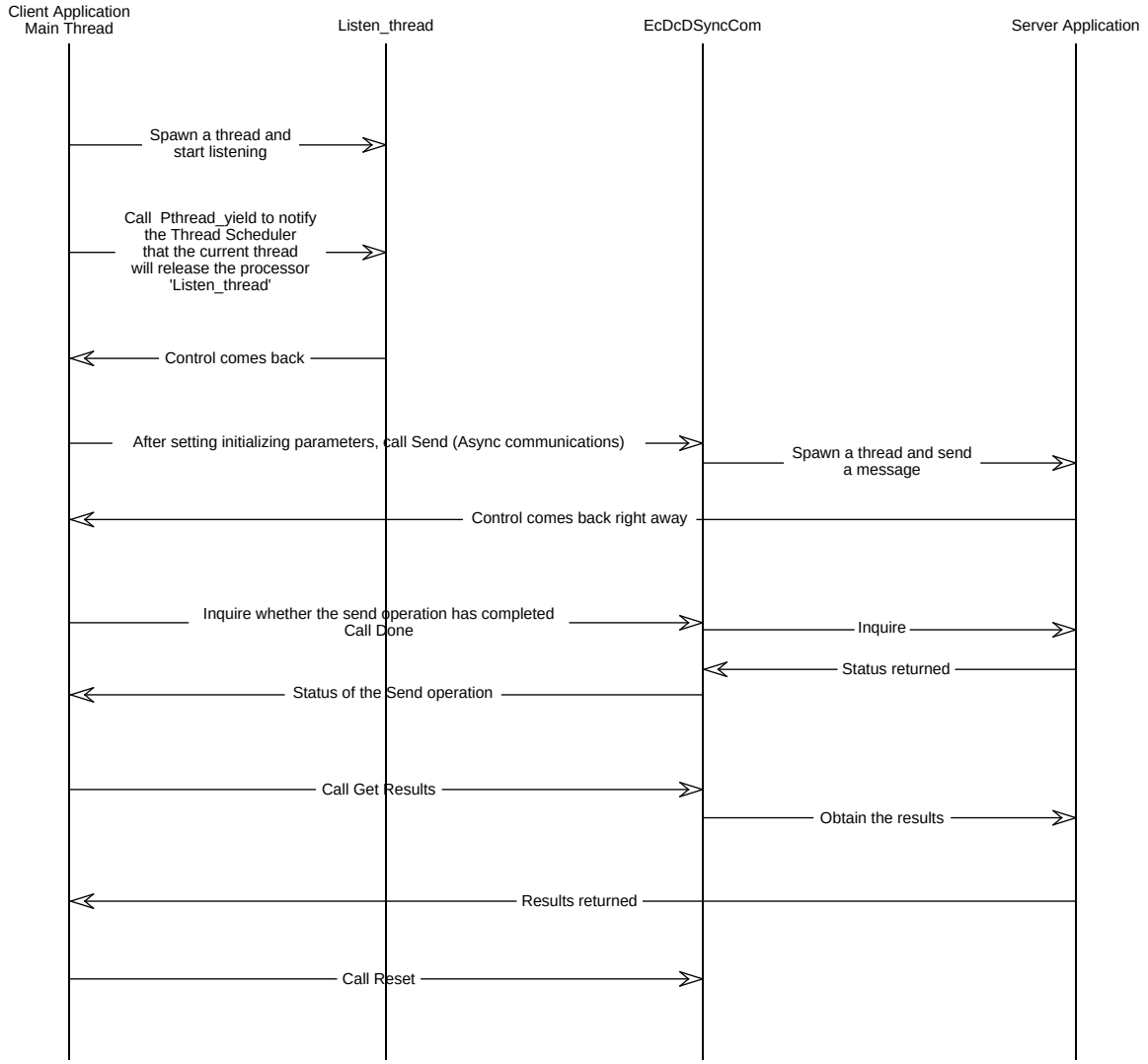

Figure 4.2.3.2.5-1. Message Passing Scenario Event Trace

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.

4.2.4.4 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 Threads 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 ReleaseB. Due to the underlying complexity of threads, only a subset of their functionality is revealed to the other segments.

Figure 4.2.5.2-1 shows, at a high level, the interaction between SDPS, FOS and MSS developers and the CSS Thread Service. Applications utilize the Thread Service transparently by setting an option when starting their server. The Thread Service will transparently cause each invocation of a server operation to run as a distinct thread.

Application developers must protect against conflicts between different threads by utilizing the Lock, TryLock and UnLock APIs as can be seen in Figure 4.2.5.2-1. The APIs synchronize the access of concurrent threads to shared data objects.

4.2.5.3 Object Model

The Thread Service Object Model in Figure 4.2.5.3-1 provides a more detailed view of the interaction possible using the Thread Service. 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, 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.

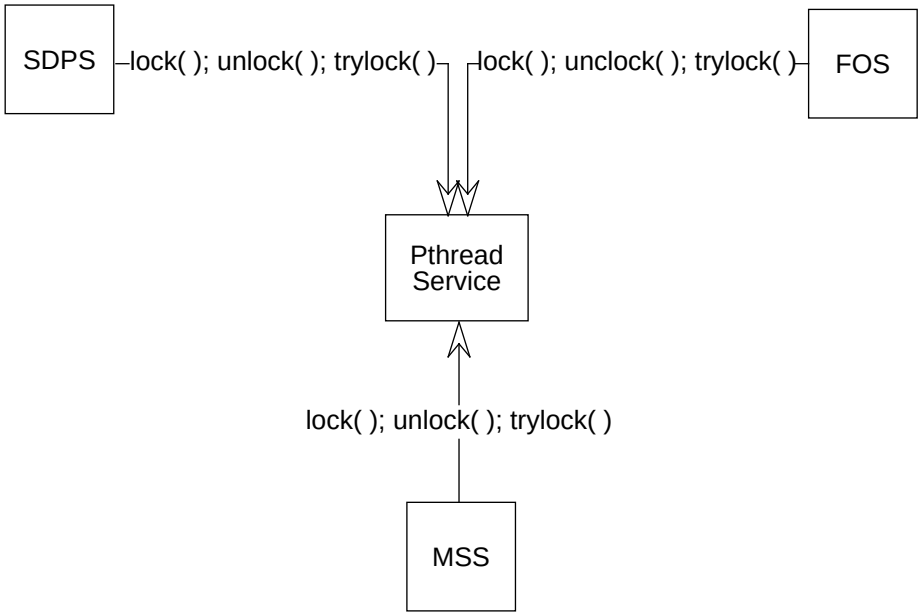

Figure 4.2.5.2-1. Thread Service Event Flow

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.

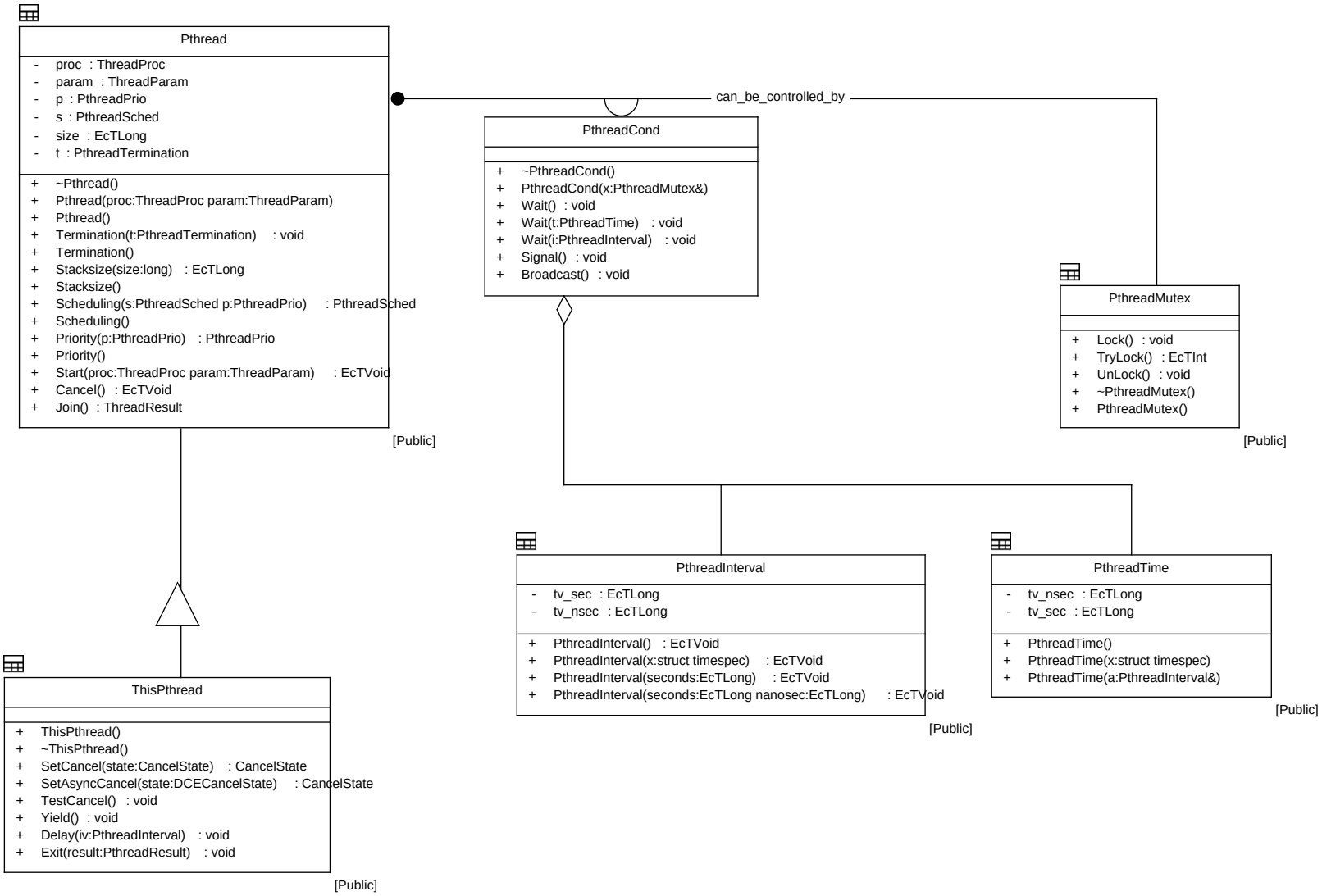

Figure 4.2.5.3-1. RelAThread_ObjectModel Object Model

4.2.5.3.1 Pthread Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:
Purpose and Description:
A Pthread is the representation of a thread of execution. A DCEPthread object corresponds to a single thread. The DCEPthread class provides limited information about a thread and limited control of that thread. A DCEPthread object represents the thread before, during, and after its execution. The thread may also continue to execute after the DCEPthread object has been deleted.

Attributes:

p - Priority to run the thread(Pthread_pri_min, Pthread_pri_low, Pthread_pri_mid, Pthread_pri_high, Pthread_pri_max).
Data Type:PthreadPrio
Privilege:Private
Default Value:Pthread_pri_mid

param - Parameter passed to thread when thread is started
Data Type:ThreadParam
Privilege:Private
Default Value:

proc - Procedure to be executed when thread is started
Data Type:ThreadProc
Privilege:Private
Default Value:
Contraints:
Non Persisent Flag:False

s - Scheduling type(Pthread_fifo- first in first out, Pthread_rr - round robin, Pthread_fg - foreground non portable, Pthread_bg - background non portable)
Data Type:PthreadSched
Privilege:Private
Default Value:Pthread_fg

size - Size of the stack for the thread
Data Type:EcTLong
Privilege:Private
Default Value:

t - Thread termination type. (Pthread_no_detach_on_delete, Pthread_detach_on_delete, or

Pthread_join_on_delete)
Data Type:PthreadTermination
Privilege:Private
Default Value:

Operations:

Cancel - Sends a cancel signal to the thread. This operation is only valid once a Pthread has been Start'ed. The thread may block cancellation, so it need not stop immediately.
Arguments:
Return Type:EcTVoid
Privilege:Public

Join - Waits for completion of thread.
Arguments:
Return Type:ThreadResult
Privilege:Public

Priority - Retrieves priority
Arguments:
Return Type:Void
Privilege:Public

Priority - Sets the priority of the thread
Arguments:p:PthreadPrio
Return Type:PthreadPrio
Privilege:Public

Pthread - Default constructor
Arguments:
Return Type:Void
Privilege:Public

Pthread - Constructors for the Pthread class which will contruct a Pthread based on default attributes. The second form will also start the thread.
Arguments:proc:ThreadProc param:ThreadParam
Return Type:Void
Privilege:Public

Scheduling - Retrieves scheduling algorithm
Arguments:
Return Type:Void
Privilege:Public

Scheduling - Sets scheduling and relative priority
Arguments:s:PthreadSched p:PthreadPrio
Return Type:PthreadSched
Privilege:Public

Stacksize - Retrieves stack size
Arguments:
Return Type:Void
Privilege:Public

Stacksize - Sets thread stack size
Arguments:size:long
Return Type:EcTLong
Privilege:Public

Start - Launches that thread by executing the specified procedure, passing it the specified parameter. This operation may only be performed on a Pthread that has no been previously Start'ed.
Arguments:proc:ThreadProc param:ThreadParam
Return Type:EcTVoid
Privilege:Public

Termination - Retrieves termination options
Arguments:
Return Type:Void
Privilege:Public

Termination - Sets termination options
Arguments:t:PthreadTermination
Return Type:void
Privilege:Public

~Pthread - The destructor cleans up the Pthread data.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The Pthread class has associations with the following classes:
Class: PthreadMutex can_be_controlled_by

4.2.5.3.2 PthreadCond Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The PthreadCond class encapsulates the pthread_cond_t condition type. A mutex is associated with the PthreadCond when it is constructed.

Attributes:

None

Operations:

Broadcast - This call signals a thread waiting on the condition variable that the condition may now be true. Broadcast wakes up all the threads that are waiting on the condition variable.
Arguments:
Return Type:void
Privilege:Public

PthreadCond - The only constructor for PthreadCond takes a mutex as a parameter. This is the mutex associated with the condition. This mutex must be locked before certain operations may take place.
Arguments:x:PthreadMutex&
Return Type:Void
Privilege:Public

Signal - This call signals a thread waiting on the condition variable that the condition may now be true. Signal wakes up one thread that is waiting on the condition variable.
Arguments:
Return Type:void
Privilege:Public

Wait - This operation initiates a wait on the condition variable. The caller should have set the mutex, then tested for the desired condition before making the call. A return implies that the mutex is now set for the caller. The condition must be re-checked before continuing. If the condition is still not true, the caller may wish to wait again.
Arguments:
Return Type:void
Privilege:Public

Wait - This operation initiates a wait on the condition variable. The caller should have set

the mutex, then tested for the desired condition before making the call. A return implies that the mutex is now set for the caller. The condition must be re-checked before continuing. If the condition is still not true, the caller may wish to wait again.
Arguments:t:PthreadTime
Return Type:void
Privilege:Public

Wait - This operation initiates a wait on the condition variable. The caller should have set the mutex, then tested for the desired condition before making the call. A return implies that the mutex is now set for the caller. The condition must be re-checked before continuing. If the condition is still true, the caller may wish to wait again.
Arguments:i:PthreadInterval
Return Type:void
Privilege:Public

~PthreadCond - The destructor deletes the Condition variable. It may only be called when there are no threads waiting on it, or on its mutex. No check is made to verify that this is the case.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The PthreadCond class has associations with the following classes:
None

4.2.5.3.3 PthreadInterval Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:
Purpose and Description:
The PthreadInterval class represents a time interval. This is used in several of the Pthread calls. Time intervals are distinguished from actual time (PthreadTime). Conversions may need to take place between time and interval, depending on the specific needs of a pthread intrinsic.

Attributes:

tv_nsec - Additional nanoseconds since tv_sec
Data Type:EcTLong
Privilege:Private

Default Value:

tv_sec - Number of seconds since 00:00:00 GMT, 1 January 1970
Data Type:EcTLong
Privilege:Private
Default Value:

Operations:

PthreadInterval - The constructor builds a time interval.
Arguments:
Return Type:EcTVoid
Privilege:Public

PthreadInterval - The constructor builds a time interval.
Arguments:x:struct timespec
Return Type:EcTVoid
Privilege:Public

PthreadInterval - The constructor builds a time interval.
Arguments:seconds:EcTLong
Return Type:EcTVoid
Privilege:Public

PthreadInterval - The constructor builds a time interval.
Arguments:seconds:EcTLong nanosec:EcTLong
Return Type:EcTVoid
Privilege:Public

Associations:

The PthreadInterval class has associations with the following classes:
PthreadCond (Aggregation)

4.2.5.3.4 PthreadMutex Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:
Purpose and Description:
The PthreadMutex class provides the fundamental locking mechanism.

Attributes:

None

Operations:

Lock - Acquires a lock on a mutex.
Arguments:
Return Type:void
Privilege:Public

PthreadMutex - Construct a mutex.
Arguments:
Return Type:Void
Privilege:Public

TryLock - Attempts to acquire a lock on a mutex.
Arguments:
Return Type:EcTInt
Privilege:Public

UnLock - Unlocks a mutex.
Arguments:
Return Type:void
Privilege:Public

~PthreadMutex - Delete the mutex. It is illegal to delete a mutex which is currently locked, but no checks are made.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The PthreadMutex class has associations with the following classes:
Class: Pthread can_be_controlled_by

4.2.5.3.5 PthreadTime Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:
Purpose and Description:

This class represents an actual time, as contrasted with a time interval represented by PthreadInterval. This class inherits timespec, as defined in pthread.h. The fields of this struct are publicly available. However, operations (including conversions) are preferable to direct use of timespec fields. The default copy constructor and assignment operator are available for use with this type.

Attributes:

tv_nsec - Additional nanoseconds since tv_sec
Data Type:EcTLong
Privilege:Private
Default Value:

tv_sec - Number of seconds since 00:00:00 GMT, 1 January 1970
Data Type:EcTLong
Privilege:Private
Default Value:

Operations:

PthreadTime - The constructor creates a PthreadTime. The timespec is initialized to the provided timespec, or to the current time plus indicated interval.
Arguments:
Return Type:Void
Privilege:Public

PthreadTime - The constructor creates a PthreadTime. The timespec is initialized to the provided timespec, or to the current time plus indicated interval.
Arguments:x:struct timespec
Return Type:Void
Privilege:Public

PthreadTime - The constructor creates a PthreadTime. The timespec is initialized to the provided timespec, or to the current time plus indicated interval.
Arguments:a:PthreadInterval&
Return Type:Void
Privilege:Public

Associations:

The PthreadTime class has associations with the following classes:
PthreadCond (Aggregation)

4.2.5.3.6 ThisPthread Class

Parent Class:Pthread
Public:Yes
Distributed Object:No
Persistent Class:
Purpose and Description:
ThisPthread is a reference to the current running thread. It is derived from Pthread. However, an object of type ThisPthread should never be referenced as a Pthread since the destructor has not been made virtual, and use of the Join and Stacksize member functions is disallowed.

Attributes:

All Attributes inherited from parent class

Operations:

Delay - Delay this thread for a period of time.
Arguments:iv:PthreadInterval
Return Type:void
Privilege:Public

Exit - Terminates execution of this thread, and sets the result that can be accessed by calling Join on this thread. Note: this function must not be called from the root thread.
Arguments:result:PthreadResult
Return Type:void
Privilege:Public

SetAsyncCancel - Control the cancellation states. CancelState is either CANCEL_OFF or CANCEL_ON.
Arguments:state:DCECancelState
Return Type:CancelState
Privilege:Public

SetCancel - Control the cancellation states. CancelState is either CANCEL_OFF or CANCEL_ON
Arguments:state:CancelState
Return Type:CancelState
Privilege:Public

TestCancel - Tests whether cancellation of the thread has occurred.
Arguments:
Return Type:void

Privilege:Public

ThisPthread - Constructor for the class
Arguments:
Return Type:Void
Privilege:Public

Yield - Allows another thread to gain control of kernel.
Arguments:
Return Type:void
Privilege:Public

~ThisPthread - Destructor for the class
Arguments:
Return Type:Void
Privilege:Public

Associations:

The ThisPthread class has associations with the following classes:
None

4.2.5.4 Dynamic Model Scenarios

4.2.5.4.1 Abstract

A process sends a message to another process. Within the message passing logic the message to be sent was placed on an outgoing queue. There are two threads processing the entries on the outgoing queue.

4.2.5.4.2 Interfaces

Thread Service

4.2.5.4.3 Stimulus

A message is being sent from one process to another process.

4.2.5.4.4 Desired Response

The message is sent successfully. The two threads processing the outgoing queue do not interfere with each other while removing the message from the queue.

4.2.5.4.5 Participating Classes

PthreadMutex

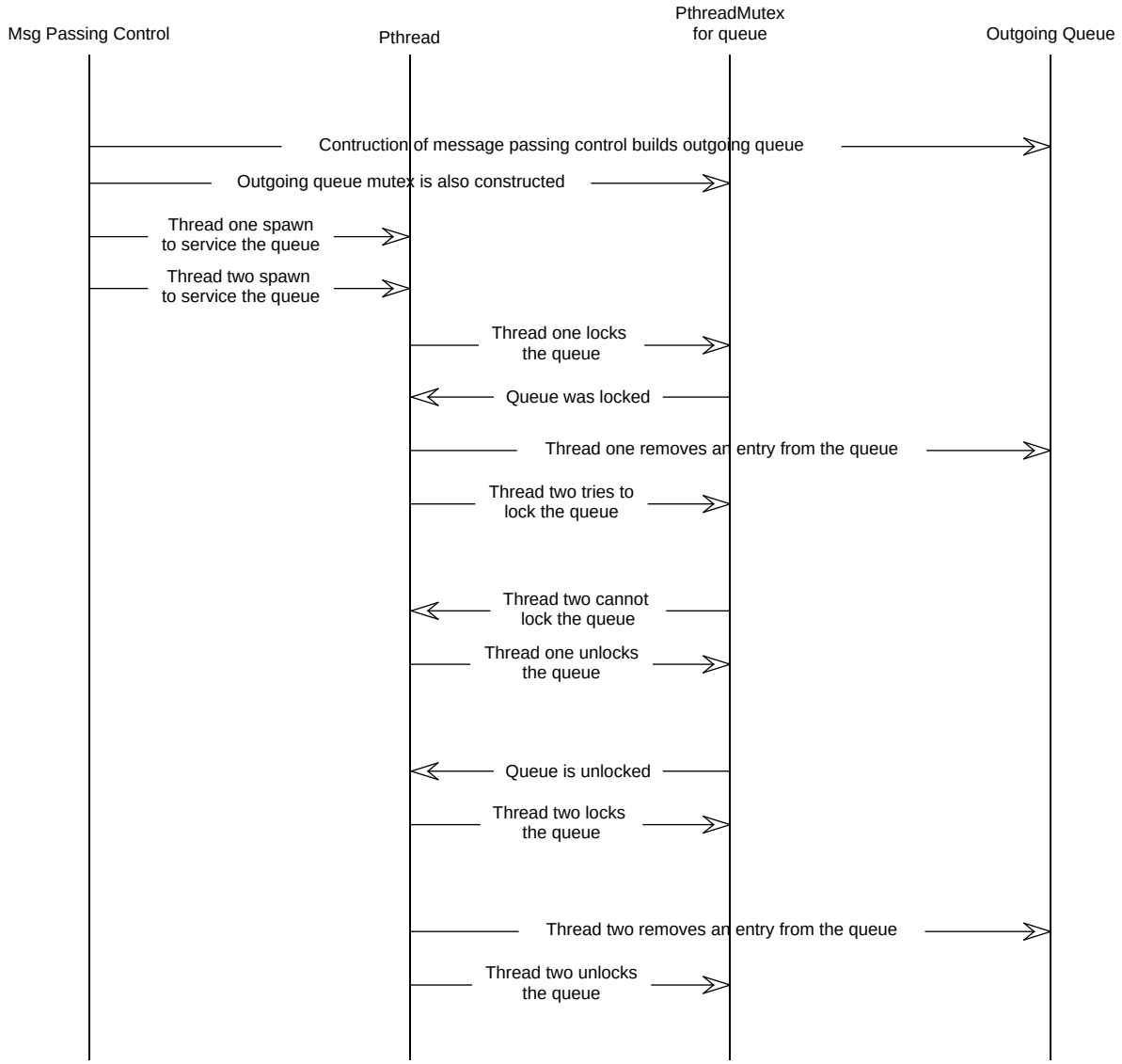

Figure 4.2.5.4-1. Thread Service Event Trace

4.2.5.4.6 Pre-conditions

The message passing control object was constructed. During construction the queues were established along with the queue locks, and threads were spawned to process the entries on the queue. A process made a request to send a message and a message was placed on the outgoing queue.

4.2.5.4.7 Post-conditions

The message on the outgoing queue was sent successfully and the message was removed from the outgoing queue.

4.2.5.4.8 Scenario Description

There is a message on the outgoing queue that is to be sent to another process/application. There are two threads processing and removing messages on/from the outgoing queue. Thread one has completed sending a message and needs to remove the message from the outgoing queue. Thread one locks the queue so that thread two does not corrupt the outgoing queue while thread one is removing the entry off the outgoing queue. Thread two tries to lock the queue but is unsuccessful. After thread one removes the entry off the outgoing queue, thread one unlocks the queue so that thread two may access the queue. Thread two can now lock the queue.

4.2.5.4.9 Event Trace

4.2.5.5 Implementation

The Thread Service is available as a COTS within the OODCE and DCE products.

4.2.5.6 Service/CSCI Management and Operation

4.2.5.6.1 Service/CSCI Management and Operation

Not Applicable.

4.2.5.6.2 System Management and Strategy

Not Applicable.

4.2.5.6.3 Reports

None.

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. Figure 4.2.6.2-1 shows, at a high level, the interaction between FOS and SCDO subsystem applications and the CSS Time Service. 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.

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

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.

Figure 4.2.6.2-1. Time Service Event Flow

4.2.6.3 Object Model

The Time Service Object Model, Figure 4.2.6.3-1, provides a more detailed view of the interaction possible using the Time Service. Time Service provides operations to obtain timestamps based on Coordinated Universal Time (UTC). The Time Service also translates different timestamp formats and performs calculations on timestamps. A basic object responsibility matrix follows:

CSS is presenting a design that provides an external feed at every DAAC, the EOC, and the isolation cell. The NASA-36 time will be received by a Time Server (product which accepts NASA-36 and converts it into UTC format) which will in turn be fed into the DTS for the EOC. All other DCE cells will require a commercial device (time provider) which can receive and interpret radio, telephone, or satellite signals to provide the UTC time values needed for DTS.

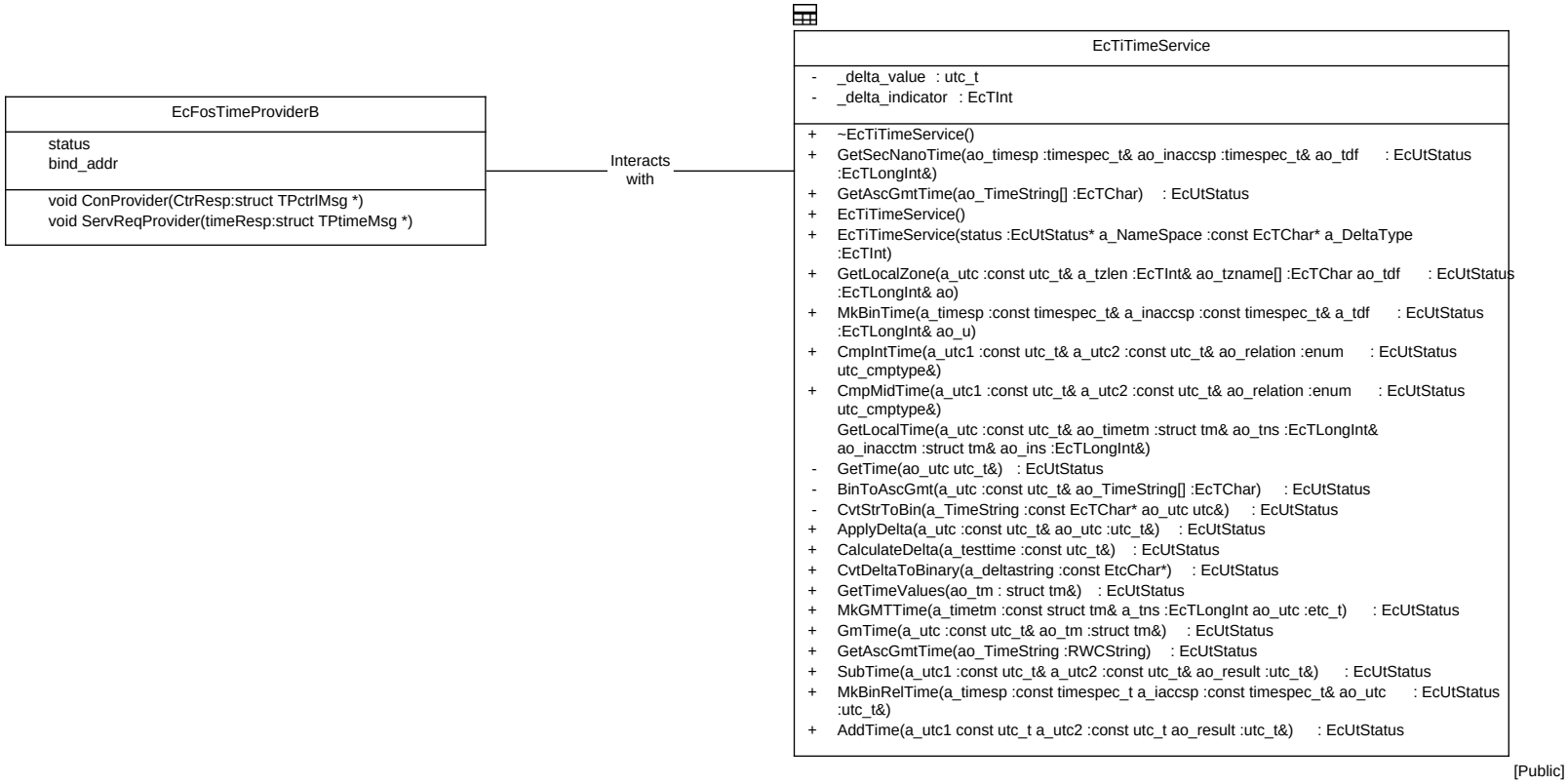

Figure 4.2.6.3-1. Time Service Object Model

Table 4.2.6.3-1. Time Service Object Responsibility Matrix

Object	Responsibility
EcTiTimeService	Retrieving timestamp information Converting between binary timestamps and ASCII representations
EcFosTimePrviderB	Responsible for extracting the time from the external time source for the FOS

4.2.6.3.1 EcFosTimeProviderB Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
This class allows a time source FOS to convert time from NASA-36 format to a given format and this outputted time can be propagated through DCE for use. The accuracy here is to a microsecond.

Attributes:

bind_addr - The handle that allows the DTS daemon to find and connect to a Time-provider process

status - The result of returned value form function call

Operations:

void ConProvider - This is the first routine called by DTS. The routine is called to verify that the TP process is running and to obtain a control message that DTS uses for subsequent communications with the TP process and for synchronization after it receives the timestamps.
Arguments:CtrResp:struct TPctrlMsg *

void ServReqProvider - After the TimeProvider is successfully contacted, DTS makes the ServerRequestProviderTime procedure call to obtain the timestamps from the external time-provider.
Arguments:timeResp:struct TPtimeMsg *

Associations:
The EcFosTimeProviderB class has associations with the following classes:
Class: EcTiTimeService Interactswith

4.2.6.3.2 EcTiTimeService Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Persistent Class:
Purpose and Description:
This class is used to obtain the current time in various formats.

Attributes:

_delta_indicator - Indicates what to do with the _delta_value; 0 = Obtain current time; 1 = Add delta to the current time.
Data Type:EcTInt
Privilege:Private
Default Value:

_delta_value - Delta that will be added to the current time.
Data Type:utc_t
Privilege:Private
Default Value:

Operations:

AddTime - This method adds two binary timestamps, producing a third binary timestamp whose inaccuracy is the sum of the two input inaccuracies.
Arguments:a_utc1 const utc_t a_utc2 :const utc_t ao_result :utc_t&
Return Type:EcUtStatus
Privilege:Public

ApplyDelta - Apply the delta to the current time
Arguments:a_utc :const utc_t& ao_utc :utc_t&
Return Type:EcUtStatus
Privilege:Public

BinToAscGmt
Arguments:a_utc :const utc_t& ao_TimeString[] :EcTChar
Return Type:EcUtStatus
Privilege:Private

CalculateDelta - Given an absolute time calculate the delta.
Arguments:a_testtime :const utc_t&
Return Type:EcUtStatus
Privilege:Public

CmpIntTime - Compares two binary timestamps or two relative binary timestamps and returns the relationship not ignoring inaccuracies.
Arguments:a_utc1 :const utc_t& a_utc2 :const utc_t& ao_relation :enum utc_cmptype&
Return Type:EcUtStatus
Privilege:Public

CmpMidTime
Arguments:a_utc1 :const utc_t& a_utc2 :const utc_t& ao_relation :enum utc_cmptype&
Return Type:EcUtStatus
Privilege:Public

CvtDeltaToBinary - Convert a delta value from the namespace to a binary timestamp.
Arguments:a_deltastring :const EtcChar*
Return Type:EcUtStatus
Privilege:Public

CvtStrToBin - Converts character string to binary timestamp.
Arguments:a_TimeString :const EcTChar* ao_utc utc&
Return Type:EcUtStatus
Privilege:Private

EcTiTimeService - Default constructor. When a NULL string is passed as a parameter in the constructor,the current time will be obtained.
Arguments:
Return Type:Void
Privilege:Public

EcTiTimeService - Constructor. Creates an instance of the class ,given a name from the namespace and a flag that indicates how delta is passed (as a relative delta or incorporated in the absolute testtime).
Arguments:status :EcUtStatus* a_NameSpace :const EcTChar* a_DeltaType :EcTInt
Return Type:Void
Privilege:Public

GetAscGmtTime - Obtain current GMT time in ASCII string format.
Arguments:ao_TimeString[] :EcTChar
Return Type:EcUtStatus
Privilege:Public

GetAscGmtTime - Obtain current GMT Time in RWCString format
Arguments:ao_TimeString :RWCString
Return Type:EcUtStatus
Privilege:Public

GetLocalTime - Converts a binary timestamp to a tm structure that expresses local time.
Arguments:a_utc :const utc_t& ao_timetm :struct tm& ao_tns :EcTLongInt& ao_inacctm :struct tm& ao_ins :EcTLongInt&

GetLocalZone - This method gets the local time zone label and offset from GMT, given utc.
Arguments:a_utc :const utc_t& a_tzlen :EcTInt& ao_tzname[] :EcTChar ao_tdf :EcTLongInt& ao
Return Type:EcUtStatus
Privilege:Public

GetSecNanoTime - Obtain current time (timespec_t - seconds, nanoseconds)
Arguments:ao_timesp :timespec_t& ao_inaccsp :timespec_t& ao_tdf :EcTLongInt&
Return Type:EcUtStatus
Privilege:Public

GetTime - Obtains current binary timestamp.
Arguments:ao_utc utc_t&
Return Type:EcUtStatus
Privilege:Private

GetTimeValues - Obtain current time (tm structure - seconds, minutes, hours, day of month, month of year, year, day of week, day of week, day of year, flag for daylight savings time).
Arguments:ao_tm : struct tm&
Return Type:EcUtStatus
Privilege:Public

GmTime - Converts a Binary stamp to a TM structure.
Arguments:a_utc :const utc_t& ao_tm :struct tm&
Return Type:EcUtStatus
Privilege:Public

MkBinRelTime - Make Binary Relative Time from the timespec_t structure
Arguments:a_timesp :const timespec_t a_iaccsp :const timespec_t& ao_utc :utc_t&
Return Type:EcUtStatus
Privilege:Public

MkBinTime - This method converts a timespec structure time to a binary timestamp.
Arguments:a_timesp :const timespec_t& a_inaccsp :const timespec_t& a_tdf :EcTLongInt& ao_u
Return Type:EcUtStatus
Privilege:Public

MkGMTTime - This method converts a trn structure that expresses GMT or UTC to a binary stamp.
Arguments:a_timetm :const struct tm& a_tns :EcTLongInt ao_utc :etc_t
Return Type:EcUtStatus
Privilege:Public

SubTime - The method subtracts one time stamp from another.
Arguments:a_utc1 :const utc_t& a_utc2 :const utc_t& ao_result :utc_t&
Return Type:EcUtStatus
Privilege:Public

~EcTiTimeService - Destructor for the EcTiTimeService Class.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcTiTimeService class has associations with the following classes:
Class: EcFosTimeProviderB Interactswith

4.2.6.4 Dynamic Model Scenarios

4.2.6.4.1 Scenario #1

4.2.6.4.1.1 Abstract

An application wishes to obtain a simulated time.

4.2.6.4.1.2 Interfaces

Time Service

4.2.6.4.1.3 Stimulus

An application constructs an EcTiTimeService object with a delta value to be used for simulated time and then makes a request for the time.

4.2.6.4.1.4 Desired Response

Application receives the correct simulated time.

4.2.6.4.1.5 Participating Classes

EcTiTimeService

4.2.6.4.1.6 Pre-conditions

The application must place an entry in the Directory Service which contains a delta value to be used in simulating time. The application must also construct an EcTiTimeService object using the "Name" in the Directory Service where the delta value can be found.

4.2.6.4.1.7 Post-conditions

The application receives the correct simulated time and continues processing.

4.2.6.4.1.8 Scenario Description

The application places an entry in the Directory Services that contains the delta value and constructs an EcTiTimeService object using the "Namespace" where the delta value was placed. The application than makes a request to obtain the ASCII GMT time.

4.2.6.4.1.9 Event Trace

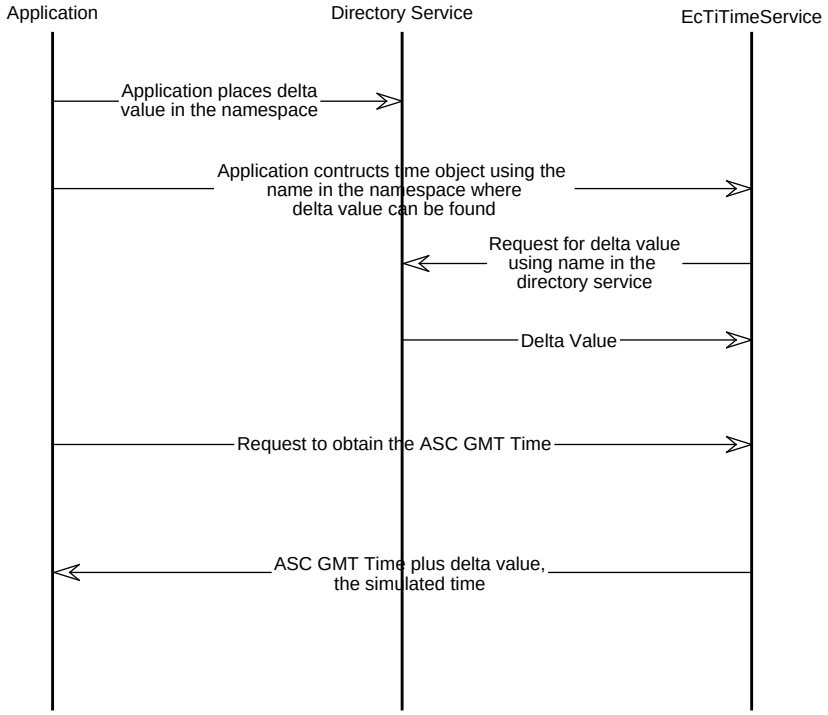

Figure 4.2.6.4-1. Time Service Event Trace

4.2.6.4.2 Scenario #2

4.2.6.4.2.1 Abstract

A FOS application wishes to convert NASA-36 Time to UTC

4.2.6.4.2.2 Interfaces

Time Provider Service

4.2.6.4.2.3 Stimulus

An application constructs EcTiTimeService object to convert time to required format.

4.2.6.4.2.4 Desired Response

Application receives desired time.

4.2.6.4.2.5 Participating Classes

EcFosTimeServiceB,EcTiTimeService

4.2.6.4.2.6 Pre-conditions

The application places constructs an EcTiTimeService object and then makes a request for the formatted time.

The EcFosTimeProvider object is instantiated and the DTS daemon is running. The system administrator has to the invoke the EcFosTimeProvider object before the DCE Time Provider Interface can extract the time from the external time source.

4.2.6.4.2.7 Post-conditions

The application receives the correct output time desired to the accuracy of a micro-second**.**

4.2.6.4.2.8 Scenario Description

The application takes the NASA-36 time provided and constructs an ECTiTimeService object. The application then uses this time provided to the DTS by the external time source.

4.2.6.4.2.9 Event Trace

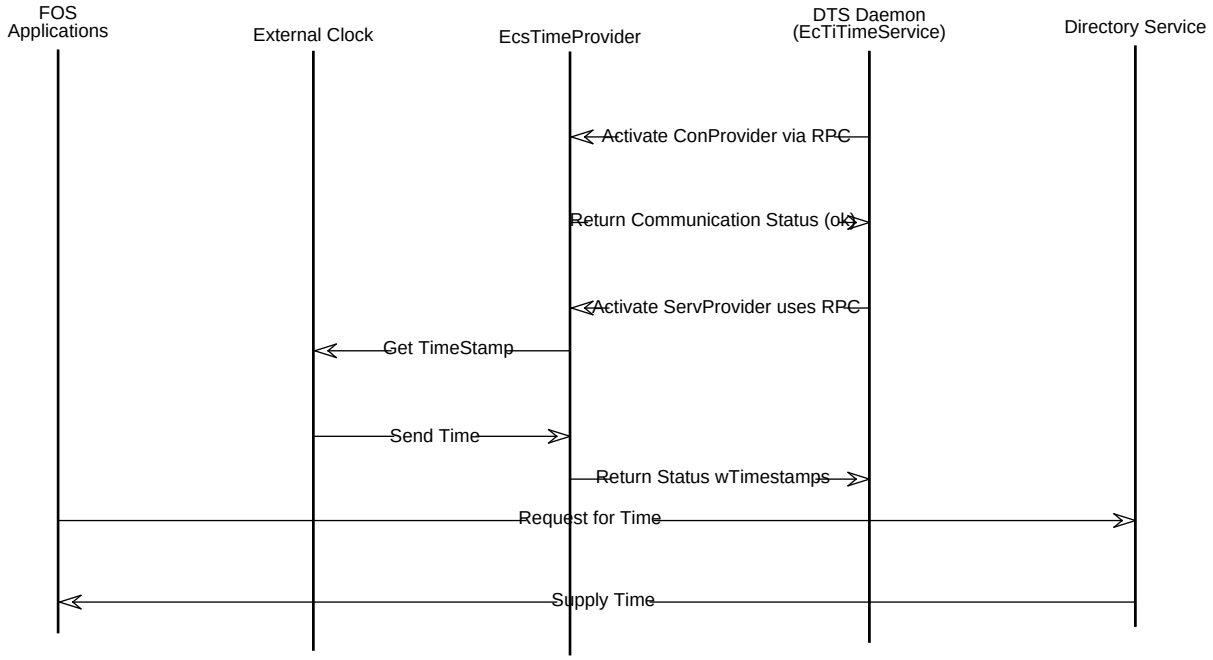

Figure 4.2.6.4-2. Time Service Event Trace

4.2.6.5 Implementation

The Time Service is available as COTS within the OODCE and DCE products. CSMS and FOS are combining requirements to procure an external Time Provider. Each DCE cell requires access to an external URL-based time provider. The resolution of DCE distributed time is millisecond accuracy. The DCE Time Services within each of the DCE cells will provide distributed time for the entire system. A FOS unique need for microsecond resolution at certain servers within the EOC will require a hardwired external source. The DCE cell at the EOC will use one of the hardwired external sources for its DCE time source.

4.2.6.6 Service/CSCI Management and Operation

4.2.6.6.1 System Management and Strategy

All system clocks have common properties that contribute to clock error and interferes with the synchronization process. If a system clock should go askew (become desynchronized) then the system clock must be synchronized with the other system clocks in the network.

4.2.6.6.2 Operator Interface

If a system clock should skew more than a specified time period away from the other system clocks in the network, operations must set the system time manually to the correct time.

4.2.6.6.3 Reports

None.

4.2.6.7 Frequently Asked Questions

1) NTP is widely used in the Internet community (and in the V0 system) and many COTS products support it, so why not use it instead of DTS?

   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 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 hardware time devices such as radios, satellites, or telephone lines, or from a software source that's connected to their cell such as NTP, which is a distributed time service on TCP/IP. 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.

2) Is DCE locked into DTS for time support? If so, why not use NTP to provide time sync, and have DTS clients on each system slaved to NTP?

One of DCE's most important features is that it is a managed environment. One of its' administrative features is that it checks for the presence of DTS servers, because of this DTS can be considered "locked into" the DCE environment. Furthermore, the system also checks that the DTS servers are accurate. Unlike other COTS products, DTS measures the inaccuracy of the distributed time. If a DTS server is found to be out of range the system will remove the server from system.

3) Ensure that host time adjustments are always made by adjusting clock rate, and never by resetting time to an earlier value.

   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. In other words, if a clock is normally incremented 10 milliseconds at each clock interrupt, and the clock is ahead, then the clock register will instead be incremented 9 milliseconds at each clock time until it reaches the correct time.

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 Object Model

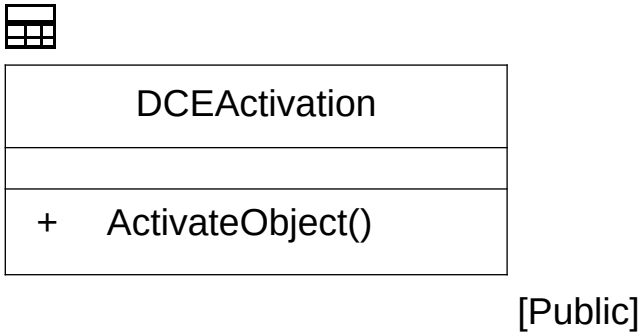

Figure 4.2.7.3-1. LifeCycle Service Object Model

4.2.7.3.1 DCEActivation Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This abstract class is used by server developers to provide an interface onto the activation of server manager objects. Since the activation of these objects is application specific, many implementations will exist. An activation object can be registered with a Server object and be used to dynamically activate manager objects.

Attributes:

None

Operations:

ActivateObject - 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.
Arguments:

Return Type:Void
Privilege:Public

Associations:

The DCEActivation class has associations with the following classes:
None

4.2.7.4 Functional Model

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 need application cooperation. For example, before suspending an application, it may be needed to bring the application to 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 exits 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 maintain 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.

4.2.7.5 Dynamic Model Scenarios

Please refer to the Scenarios # 5 and #6 in DOF.

4.2.7.6 Implementation

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

4.2.7.7 Service/CSCI Management and Operation

4.2.7.7.1 System Management and Strategy

Not Applicable.

4.2.7.7.2 Operator Interface

Not Applicable.

4.2.7.7.3 Reports

None.

4.2.8 Generic Security Service

4.2.8.1 Overview

Closely linked, conceptually, to the security service, the GSS (Generic Security Service) provides security to clients outside of the DCE realm. The GSS is a feature of DCE 1.1 which allows clients outside of a DCE cell to be authenticated by DCE security. The GSS also allows clients outside the DCE realm to communicate securely with servers inside the DCE realm. The GSS does not rely on DCE RPCs as means of communication nor does it specify any particular communication mechanism to use. The programmer must supply the communication routines to the GSS. Since the GSS APIs provided by DCE are complicated, difficult to use and not based on an object oriented design, two classes (EcSeGSSB and EcSeGSSTCPB) have been created to alleviate these problems. Use of the EcSeGSSB class is fairly simple. Note that EcSeGSSB is an abstract class and cannot be instantiated. A subclass, EcSeGSSTCPB, is provided for those who will be communicating via TCP sockets. If some other protocol is to be used, the programmer must create their own subclass of EcSeGSSB. The EcSeGSSB class provides a number of functions for use by the programmer. The InitSecContext and AcceptSecContext functions are used to establish a security context. Once a security context is established the client and server are authenticated with each other. All communication thereafter is authenticated. The SendData and RcvData functions allow for data to be sent using no security, data integrity or data privacy. The InitSecContext, AcceptSecContext, SendData and RcvData functions all use the virtual ReadData and WriteData functions for transmitting information. The SecureMsg and UnSecureMsg functions do the same thing as the SendData and RcvData without actually transmitting data. Instead an encrypted buffer is returned to be sent explicitly by the process. This feature allows for easier incorporation with legacy code and for instances where the data must be sent by means other than ReadData and WriteData. When the security context is no longer needed, simply delete the EcSeGSSTCPB object.

<u>Subclassing EcSeGSSB</u>

Two functions must be written in order to subclass EcSeGSSB: ReadData and WriteData. These are the pure virtual functions that make EcSeGSSB abstract. A constructor for the subclass must also be created even if it does nothing but call one of the EcSeGSSB constructors.

<u>Using the EcSeGSSTCPB class</u>

Three constructors are provided for instantiating the class. If the default constructor is used, a security context will not be created and either InitSecContext or AcceptSecContext must be called to establish the security context. If one of the other constructors is used a security context will be created upon instantiation and neither InitSecContext nor AcceptSecContext should be called. The programmer should never call both InitSecContext and AcceptSecContext. InitSecContext and the corresponding constructor are for use only by the client to initiate the security context. AcceptSecContext and its corresponding constructor are for use only by the server to accept a security context. Once a security context has been established the programmer may proceed to call the other EcSeGSSB member functions as necessary. The EcSeGSSB class was inherited publicly so all public member functions are available to EcSeGSSTCPB.

4.2.8.2 Context

The GSSAPI allows non-DCE RPC applications to take advantage of the DCE Security services for authentication. ECS supports the GSS for applications which wish to interact securely with the ECS system but can not take advantage of DCE RPCs. Such applications include legacy systems and the gateway.

4.2.8.3 Object Model

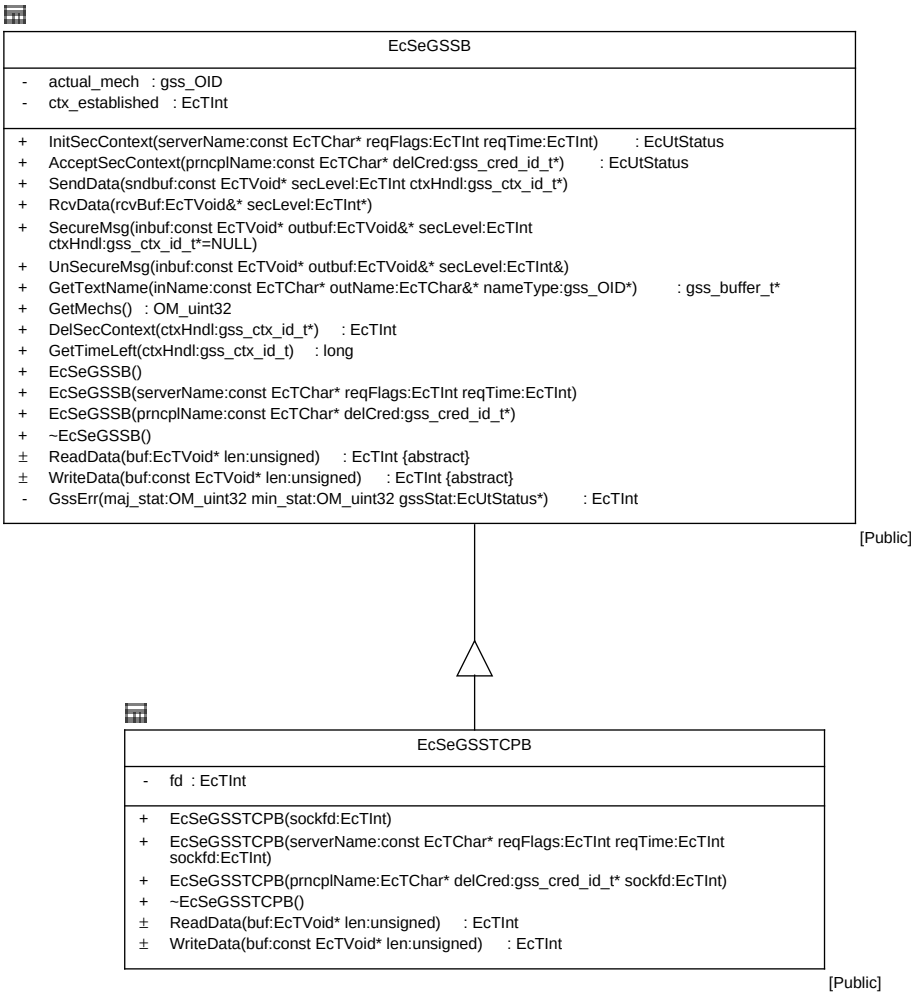

Figure 4.2.8.3-1. GSS Object Model

4.2.8.3.1 EcSeGSSB Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This is an abstract class which provides the bulk of the functionality for GSS. A user of

this class must derive a class from this one which implements the ReadData and WriteData member functions at a minimum. The EcSeGSSTCPB class is an example of such a derivation and the default class which most people should use.

Attributes:

actual_mech - The underlying mechanism used for authentication. May be either DCE or Kerberos.
Data Type:gss_OID
Privilege:Private
Default Value:

ctx_established - Indicates whether or not a security context has been established. This attribute is checked by all member functions that require a previously established context. The function will fail if this attribute is not set.
Data Type:EcTInt
Privilege:Private
Default Value:

Operations:

AcceptSecContext - This function is called only by the server process and serves to accept a security context initiated by a client. This must be the first GSS member function called unless the corresponding constructor is used (see constructors).
Arguments:prncplName:const EcTChar* delCred:gss_cred_id_t*
Return Type:EcUtStatus
Privilege:Public

DelSecContext - This functioin deletes the security context associated with the object. Once this function is called none of the other member functions will work except InitSecContext and AcceptSecContext.
Arguments:ctxHndl:gss_ctx_id_t*
Return Type:EcTInt
Privilege:Public

EcSeGSSB - Default constructor - This allows a GSS object to be created without establishing a security context. Note, however, that a security context must be established before any of the GSS may be used (see InitSecContext and AcceptSecContext).
Arguments:
Return Type:Void
Privilege:Public

EcSeGSSB - Client Constructor - This constructor is used only by the client when the client wishes to establish a security context upon the creation of the GSS object. If this

constructor is used, the InitSecContext function should not be called.
Arguments:serverName:const EcTChar* reqFlags:EcTInt reqTime:EcTInt
Return Type:Void
Privilege:Public

EcSeGSSB - Server Constructor - This constructor is used only by the server when the server wishes to establish a security context upon the creation of the GSS object. If this constructor is used the AcceptSecContext function should not be called.
Arguments:prncplName:const EcTChar* delCred:gss_cred_id_t*
Return Type:Void
Privilege:Public

GetMechs - Returns the value of the actual_mech attribute.
Arguments:
Return Type:OM_uint32
Privilege:Public

GetTextName - Returns a textual representation of an opaque internal name.
Arguments:inName:const EcTChar* outName:EcTChar&* nameType:gss_OID*
Return Type:gss_buffer_t*
Privilege:Public

GetTimeLeft - Returns the remaining amount of time for which the security context is established.
Arguments:ctxHndl:gss_ctx_id_t
Return Type:long
Privilege:Public

GssErr - This function determines if a DCE GSS error occured and if so, convertes it to an EcUtStatus.
Arguments:maj_stat:OM_uint32 min_stat:OM_uint32 gssStat:EcUtStatus*
Return Type:EcTInt
Privilege:Private

InitSecContext - This function is called by the client process only and serves to initiate the establishment of a security context. This must be the first GSS member function called unless the corresponding constructor is used (see constructors).
Arguments:serverName:const EcTChar* reqFlags:EcTInt reqTime:EcTInt
Return Type:EcUtStatus
Privilege:Public

RcvData - This function will receive a buffer using the virtual ReadData member function, unsecure it and return both the buffer and the security level used to secure the message.
Arguments:rcvBuf:EcTVoid&* secLevel:EcTInt*
Return Type:Void

Privilege:Public

ReadData - This is a pure virtual member function for reading a buffer from some communication device. This function is used by InitSecContext, AcceptSecContext, SendData and RcvData.
Arguments:buf:EcTVoid* len:unsigned
Return Type:EcTInt
Privilege:Protected
This is an abstract operation

SecureMsg - This function applies security to an input buffer and returns an opaque version of the buffer without sending the buffer. This function is good for incorporating GSS into existing code and for cases where data must be sent via some means other than the WriteData member function.
Arguments:inbuf:const EcTVoid* outbuf:EcTVoid&* secLevel:EcTInt ctxHndl:gss_ctx_id_t*=NULL
Return Type:Void
Privilege:Public

SendData - This function will apply the specified secuity to the given buffer and send the buffer using the virtual WriteData member function.
Arguments:sndbuf:const EcTVoid* secLevel:EcTInt ctxHndl:gss_ctx_id_t*
Return Type:Void
Privilege:Public

UnSecureMsg - This function takes an opaque buffer, unsecures it and returns both the readable buffer and the security level used to secure it. This function is good for incorporating GSS into existing code and for instances where data must be received via some means other that the ReadData member function.
Arguments:inbuf:const EcTVoid* outbuf:EcTVoid&* secLevel:EcTInt&
Return Type:Void
Privilege:Public

WriteData - This is a pure virtual function for writing a buffer to some communication device. This function is used by InitSecContext, AcceptSecContext, SendData, RcvData.
Arguments:buf:const EcTVoid* len:unsigned
Return Type:EcTInt
Privilege:Protected
This is an abstract operation

~EcSeGSSB - Destructor deletes the security context and destroys the object.
Arguments:
Return Type:Void
Privilege:Public

Associations:
The EcSeGSSB class has associations with the following classes:
None

4.2.8.3.2 EcSeGSSTCPB Class

Parent Class:EcSeGSSB
Public:Yes
Distributed Object:No
Purpose and Description:
This is the concrete derivation of the EcSeGSSB class. This class implements the GSS using TCP sockets. A connection must be established prior the instantiating this object.

Attributes:

fd - This is the socket file descriptor associated with the connection that this object is to use.
Data Type:EcTInt
Privilege:Private
Default Value:

Operations:

EcSeGSSTCPB - This constructor instantiates the class using a socket file descriptor from a connection that must already be established. I also calls the default constructor for EcSeGSSB.
Arguments:sockfd:EcTInt
Return Type:Void
Privilege:Public

EcSeGSSTCPB - This constructor initializes this class with the socket file descriptor and then calls the corresponding EcSeGSSB constructor.
Arguments:serverName:const EcTChar* reqFlags:EcTInt reqTime:EcTInt sockfd:EcTInt
Return Type:Void
Privilege:Public

EcSeGSSTCPB - This constructor initializes the socket file descriptor and then calls the corresponding EcSeGSSB constructor.
Arguments:prncplName:EcTChar* delCred:gss_cred_id_t* sockfd:EcTInt
Return Type:Void
Privilege:Public

ReadData - This is the TCP socket implementation of the pure virtual ReadData function in EcSeGSSB.
Arguments:buf:EcTVoid* len:unsigned

Return Type:EcTInt
Privilege:Protected

WriteData - This the TCP socket implementation of the pure virtual WriteData function from EcSeGSSB.
Arguments:buf:const EcTVoid* len:unsigned
Return Type:EcTInt
Privilege:Protected

~EcSeGSSTCPB - Calls the EcSeGSSB destructor then destroys the object.
Arguments:
Return Type:Void
Privilege:Public

Associations:
The EcSeGSSTCPB class has associations with the following classes:
None

4.2.8.4 Dynamic Model Scenarios

4.2.8.4.1 Scenario #1

4.2.8.4.1.1 Abstract

This scenario traces the interactions of a client with the class EcSeGSSTCPB.

4.2.8.4.1.2 Interfaces

Custom application.

4.2.8.4.1.3 Stimulus

Client invokes InitSecContext either explicitly or via the constructor.

4.2.8.4.1.4 Desired Response

The client establishes security context with server and sends a secure message.

4.2.8.4.1.5 Participating Classes

EcSeGSSTCPB

4.2.8.4.1.6 Pre-conditions

A TCP socket connection is established between the client and the server.

4.2.8.4.1.7 Post-conditions

None

4.2.8.4.1.8 Scenario Description

The client instantiates the EcSeGSSTCPB object then initiates the creation of a security context by calling InitSecContext. The InitSecContext routine will pass information among the DCE Secrutiy Server and the server application and eventually return with a security context. Once the client has the security context the client is authenticated with the server. The client, wishing to send a secure message to the server, calls SecureMsg with the message data to be sent. The SecureMsg function returns an opaque encrypted data buffer to the client. The client then sends the data buffer to the server.

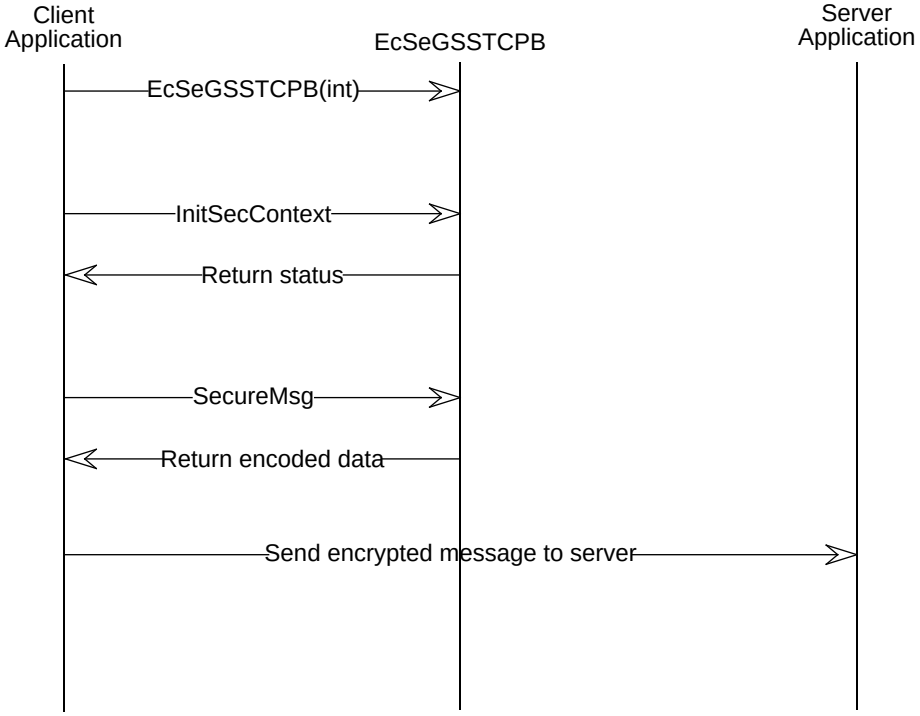

Figure 4.2.8.4.1-1. GSS Client Scenario Event Trace

4.2.8.4.2 Scenario #2

4.2.8.4.2.1 Abstract

This scenario traces the interactions of a server with the class EcSeGSSTCPB.

4.2.8.4.2.2 Interfaces

Custom application.

4.2.8.4.2.3 Stimulus

Server invokes AcceptSecContext either explicitly or via the constructor.

4.2.8.4.2.4 Desired Response

The server establishes security context with client and receives a secure message.

4.2.8.4.2.5 Participating Classes

EcSeGSSTCPB

4.2.8.4.2.6 Pre-conditions

A TCP socket connection is established between the client and the server.

4.2.8.4.2.7 Post-conditions

None

4.2.8.4.2.8 Scenario Description

The server instantiates the EcSeGSSTCPB object then calls AcceptSecContext. The AcceptSecContext routine will pass information among the DCE Secrutiy Server and the client application and eventually return with a security context. Once the server has the security context the it is authenticated with the client. When the server receives an encrypted data buffer from the client it calls UnSecureMsg with the data buffer. The UnSecureMsg function returns with the original, unencrypted message.

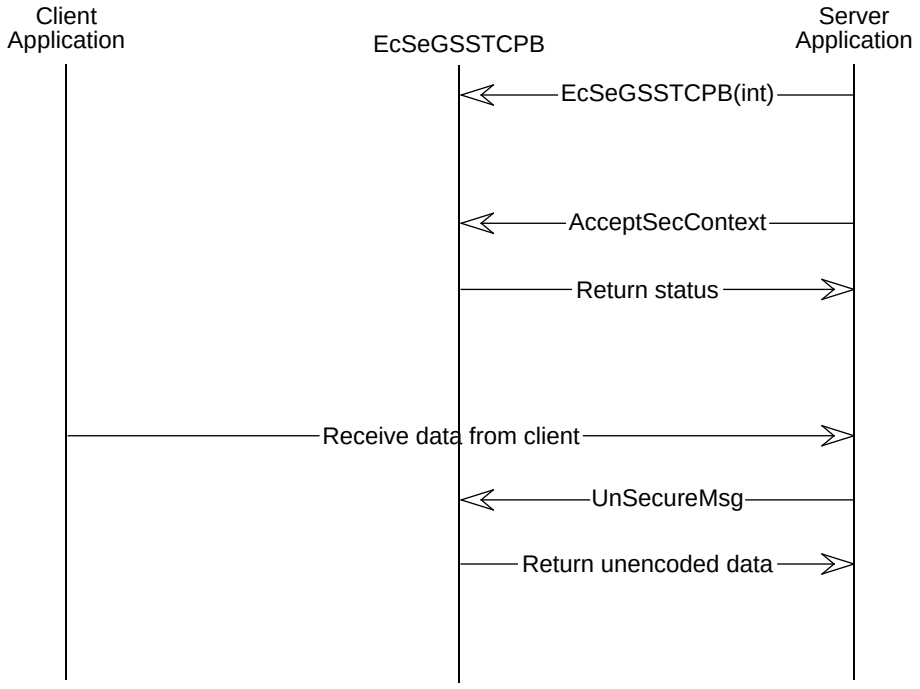

Figure 4.2.8.4.2-1. GSS Server Scenario Event Trace

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 unmarshalling. 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.

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 in coming calls and redirect them to the appropriate implementation object, create/maintain Access Control Lists associated with methods defined in the interface.

Figure 4.3.1-1. Distributed Object Framework Event Flow (1 of 3)

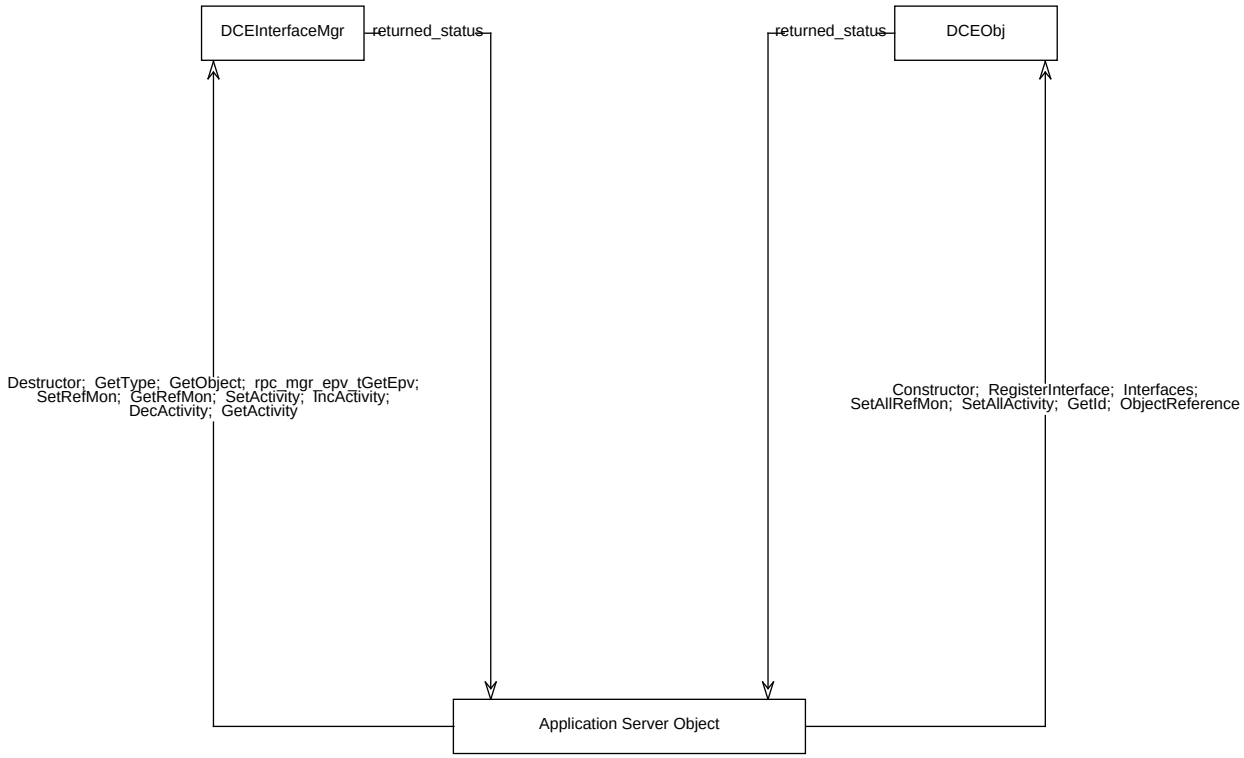

Figure 4.3.1-1. Distributed Object Framework Event Flow (2 of 3)

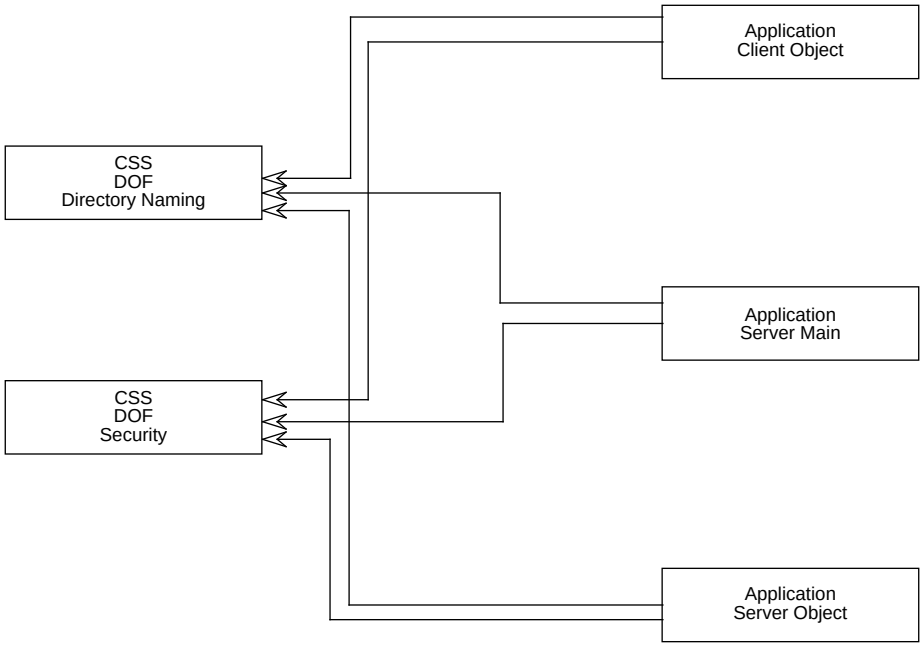

Figure 4.3.1-1. Distributed Object Framework Event Flow (3 of 3)

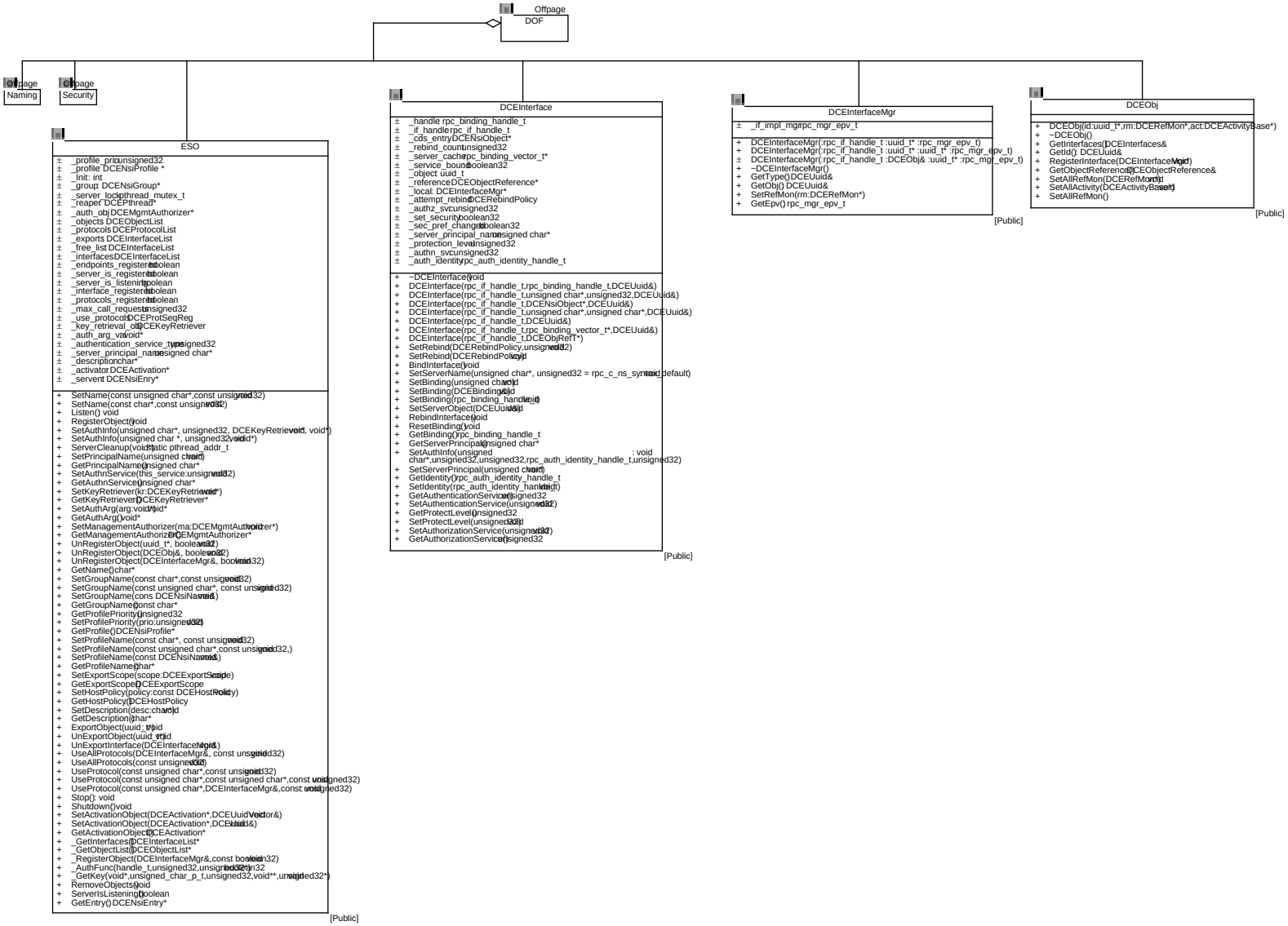

Figure 4.3.3-1. Distributed Object Framework Object Model

4.3.3.1 DCEInterface Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
The application client class inherits from the DCEInterface class which provides the default functionality required at the client side. It includes locating a service, binding and accessing the remote objects managed by that server. Client class inherits from this DCEInterface class and as such this interface is embedded in the client class. The application programmer can modify the default behavior provided by the DCEInterface class to achieve any needed special behavior.

Attributes:

_attempt_rebind - rebind policy
Data Type:DCERebindPolicy
Privilege:Protected
Default Value:

_auth_identity - Identity
Data Type:rpc_auth_identity_handle_t
Privilege:Protected
Default Value:

_authn_svc - Authentication
Data Type:unsigned32
Privilege:Protected
Default Value:

_authz_svc - Authorization
Data Type:unsigned32
Privilege:Protected
Default Value:

_cds_entry - CDS name
Data Type:DCENsiObject*
Privilege:Protected
Default Value:

_handle - Binding handle
Data Type:rpc_binding_handle_t
Privilege:Protected
Default Value:

_if_handle - DCEInterface
Data Type:rpc_if_handle_t
Privilege:Protected
Default Value:

_local - local object
Data Type:DCEInterfaceMgr*
Privilege:Protected
Default Value:

_object - Object UUID
Data Type:uuid_t
Privilege:Protected
Default Value:

_protection_level - Data security
Data Type:unsigned32
Privilege:Protected
Default Value:

_rebind_count - Number of attempts to make
Data Type:unsigned32
Privilege:Protected
Default Value:

_reference - object reference
Data Type:DCEObjectReference*
Privilege:Protected
Default Value:

_sec_pref_changed - Flag
Data Type:boolean32
Privilege:Protected
Default Value:

_server_cache - DCEBinding cache
Data Type:rpc_binding_vector_t*
Privilege:Protected
Default Value:

_server_principal_name - DCEServer principal
Data Type:unsigned char*
Privilege:Protected
Default Value:

_service_bound - Flag
Data Type:boolean32
Privilege:Protected
Default Value:

_set_security - Flag
Data Type:boolean32
Privilege:Protected
Default Value:

Operations:

BindInterface - Used to obtain a fully bound handle from the location and binding information held in the internal state of the object. This memeber function ensures that a compatible Server object is available and is able to support the Interface and object requirements of the client object. The BindInterface member function only needs to be used by the application code if fine grain control is requred over the binding of clients to servers. If the client object has not been bound to a server when the first RPC call is made, the BindInterface function is called has no effect if called when a valid binding exists. The Binding Interface member is virtual so that the binding policy may be overloaded in derived classes that require specialized binding mechanism.
Arguments:
Return Type:void
Privilege:Public

DCEInterface - Constructor. Construct from Object reference.
Arguments:rpc_if_handle_t,DCEObjRefT*
Return Type:Void
Privilege:Public

DCEInterface - Constructor. Construct from DCEUuid. If DCEUuid& null, use the once given.
Arguments:rpc_if_handle_t,DCEUuid&
Return Type:Void
Privilege:Public

DCEInterface - Constructor. Construct from binding vector and DCEUuid
Arguments:rpc_if_handle_t,rpc_binding_vector_t*,DCEUuid&
Return Type:Void
Privilege:Public

DCEInterface - Constructor. Construct from hostname/protocol
Arguments:rpc_if_handle_t,unsigned char*,unsigned char*,DCEUuid&
Return Type:Void

Privilege:Public

DCEInterface - Constructor. Construct from DCENsiObject
Arguments:rpc_if_handle_t,DCENsiObject*,DCEUuid&
Return Type:Void
Privilege:Public

DCEInterface - Constructor. Construct from unsigned string
Arguments:rpc_if_handle_t,unsigned char*,unsigned32,DCEUuid&
Return Type:Void
Privilege:Public

DCEInterface - Constructor. Construct from a binding
Arguments:rpc_if_handle_t,rpc_binding_handle_t,DCEUuid&
Return Type:Void
Privilege:Public

GetAuthenticationService - Used to get the authentication service type required by the client to perform authentication.
Arguments:
Return Type:unsigned32
Privilege:Public

GetAuthorizationService - Used to get the client side identity for the communication with a Server object.
Arguments:
Return Type:unsigned32
Privilege:Public

GetBinding - Returns the current binding information for the interface object. It returns information from internal state and as such the return value is marked as const.
Arguments:
Return Type:rpc_binding_handle_t
Privilege:Public

GetIdentity - Used to get the client side identity for the communication with a Server object.
Arguments:
Return Type:rpc_auth_identity_handle_t
Privilege:Public

GetProtectLevel - Used to get required data protection levels between the client and the server.
Arguments:
Return Type:unsigned32

Privilege:Public

GetServerPrincipal - Used to get the principal nameof the server object.
Arguments:
Return Type:unsigned char*
Privilege:Public

RebindInterface - Rebinds the binding information and perform checks that the Server object can support the client object requirements.
Arguments:
Return Type:void
Privilege:Public

ResetBinding - Resets endpoint information contained in the binding handle. The client object will re-bind the endpoint information on the next RPC call.
Arguments:
Return Type:void
Privilege:Public

SetAuthInfo - Used to set the client side security preferences.
Arguments:unsigned
char*,unsigned32,unsigned32,rpc_auth_identity_handle_t,unsigned32
Return Type:void
Privilege:Public

SetAuthenticationService - Set the authentication service type required by the client to perform authentication.
Arguments:unsigned32
Return Type:void
Privilege:Public

SetAuthorizationService - Used to set the client side identity for the communication with a Server object.
Arguments:unsigned32
Return Type:void
Privilege:Public

SetBinding - Resets the binding information held in the internal state of the object. Any existing binding information is removed and new binding checks are performed on the next RPC call. It takes a binding handle as an argument.
Arguments:rpc_binding_handle_t
Return Type:void
Privilege:Public

SetBinding - Resets the binding information held in the internal state of the object. Any existing binding information is removed and new binding checks are performed on the next RPC call. It takes a DCEBinding as an argument.
Arguments:DCEBinding&
Return Type:void
Privilege:Public

SetBinding - Resets the binding information held in the internal state of the object. Any existing binding information is removed and new binding checks are performed on the next RPC call. It takes an string as argument.
Arguments:unsigned char*
Return Type:void
Privilege:Public

SetIdentity - Used to set the client side identity for the communication with a Server object.
Arguments:rpc_auth_identity_hanlde_t
Return Type:void
Privilege:Public

SetProtectLevel - Used to set required data protection levels between the client and the server.
Arguments:unsigned32
Return Type:void
Privilege:Public

SetRebind - Sets the rebind policy of the interface object. By default the interface object has the rebind policy of never_rebind. The rebind policy allows the specification of a re-bind policy for client objects. If communication with a Server object results in a failure the client object can be instructed to attempt a re-bind. The rebind policy can be: 1.never_rebind:No rebind is done, failure is passed reported to caller as an exception. 2.attempt_rebind:Client object will attempt to rebind and replay the failed operation. The default re-bind 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 object). Re-bind is attempted once if the rebind or the re-playing of the failed operation presents another failure and an exception is passed to the client.
3.wait_on_rebind: Does the same as the attempt_rebind option, however this option will continue to retry a re-bind until it is successful or the calling thread is terminated. It takes as argument the DCERebindPolicy.
Arguments:DCERebindPolicy
Return Type:void
Privilege:Public

SetRebind - Sets the rebind policy of the interface object. By default the interface object has the rebind policy of never_rebind. The rebind policy allows the specification of a re-bind policy for client objects. If communication with a Server object results in a failure the client object can be instructed to attempt a re-bind. The rebind policy can be: 1.never_rebind:No rebind is done, failure is passed reported to caller as an exception. 2.attempt_rebind:Client object will attempt to rebind and replay the failed operation. The default re-bind 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 object). Re-bind is attempted once if the rebind or the re-playing of the failed operation presents another failure and an exception is passed to the client.
3.wait_on_rebind: Does the same as the attempt_rebind option, however this option will continue to retry a re-bind until it is successful or the calling thread is terminated. It takes as arguments the DCERebindPolicy and an integer.
Arguments:DCERebindPolicy,unsigned32
Return Type:void
Privilege:Public

SetServerName - Set the CDS entry name that is used to locate the required Server object. When SetServerName is called, the existing binding handle is invalidated and the client object will rebind using the new server name on the next RPC call.
Arguments:unsigned char*, unsigned32 = rpc_c_ns_syntax_default
Return Type:void
Privilege:Public

SetServerObject - Sets or modifies the DCE object uuid of a client binding. Checks are made to ensure the Server Object supports the required DCE object uuid.
Arguments:DCEUuid&
Return Type:void
Privilege:Public

SetServerPrincipal - Used to set the principal name of the server object.
Arguments:unsigned char*
Return Type:void
Privilege:Public

~DCEInterface - Destructor
Arguments:
Return Type:void
Privilege:Public

Associations:

The DCEInterface class has associations with the following classes:
DOF (Aggregation)

4.3.3.2 DCEInterfaceMgr Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
There can be multiple implementations for a given interface. The application server implementation class inherits from the InterfaceMgr class and the DCEObj class. The DCEInterfaceMgr class is the base abstract class from which all generated server side classes are derived. This class allows the application programmer to define and attach a RefMon class where the desired level of security (authentication) can be specified. The DCEObj class provides the concept of the Generic object which can have multiple interfaces. This class provides the functionality to associate a RefMon object to all the interfaces that it is associated to. Server implementation class inherits from these classes and as such their interfaces are embedded in the server class. The application programmer can modify the default behavior provided by these classes to achieve any needed special behavior.

Attributes:

_if_impl_mgr - C Manager epv for DCE
Data Type:rpc_mgr_epv_t
Privilege:Protected
Default Value:

Operations:

DCEInterfaceMgr - This constructor creates a DCEInterface object with the interface identifier.
Arguments::rpc_if_handle_t :uuid_t* :rpc_mgr_epv_t
Return Type:Void
Privilege:Public

DCEInterfaceMgr - This constructor will create DCEInterfaceMgr object with the interface and the object identifier.
Arguments::rpc_if_handle_t :uuid_t* :uuid_t* :rpc_mgr_epv_t
Return Type:Void
Privilege:Public

DCEInterfaceMgr - This constructor creates DCEInterfaceMgr object with the object reference.
Arguments::rpc_if_handle_t :DCEObj& :uuid_t* :rpc_mgr_epv_t
Return Type:Void

Privilege:Protected

GetEpv - Used to access the endpoint vector for the DCEInterfaceMgr object. Access to this data structure is required by the class library but not normally needed for general development.
Arguments:
Return Type:rpc_mgr_epv_t
Privilege:Public

GetObj - Provides access to the DCE object identifier information for the DCEInterfaceMgr object.
Arguments:
Return Type:DCEUuid&
Privilege:Public

GetType - Provides access to the DCE implementation type information for the DCEInterfaceMgr object.
Arguments:
Return Type:DCEUuid&
Privilege:Public

SetRefMon - Set reference monitor
Arguments:rm:DCERefMon*
Return Type:Void
Privilege:Public

~DCEInterfaceMgr - This is the destructor. It will deallocate memory occupied by DCEInterfaceMgr object.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The DCEInterfaceMgr class has associations with the following classes:
DOF (Aggregation)

4.3.3.3 DCEObj Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class provides the concept of DCE object which is a logical entity and 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.

Attributes:

None

Operations:

DCEObj - Constructor.
Arguments:id:uuid_t*,rm:DCERefMon*,act:DCEActivityBase*
Return Type:Void
Privilege:Public

GetId - Obtains the object UUID associated with the DCEObj class instance.
Arguments:
Return Type:DCEUuid&
Privilege:Public

GetInterfaces - Obtains a list of the interfaces supported by the manager object.
Arguments:
Return Type:DCEInterfaces&
Privilege:Public

GetObjectReference - Obtains an object reference fro the DCEObj class instance. This reference can be used by clients to create a binding to the server base object.
Arguments:
Return Type:DCEObjectReference&
Privilege:Public

RegisterInterface - Registers the interface data with the DCEObj class. This class maintains a list of all interfaces supported by the manager object associated with the DCEObj class.
Arguments:DCEInterfaceMgr*
Return Type:void
Privilege:Public

SetAllActivity - Sets the activity object for all of the DCEInterfaceMgr objects registered with DCEObj. This allows multiple objects to share the same activity object and counter.
Arguments:DCEActivityBase*
Return Type:void
Privilege:Public

SetAllRefMon - Used to associate a RefMon object with all of the DCEInterfaceMgr objects registered with the DCEOjec object. Therefore, all interfaces supported by the

manager object share the same RefMon object.
Arguments:
Return Type:Void
Privilege:Public

SetAllRefMon - Set reference monitor
Arguments:DCERefMon*
Return Type:void
Privilege:Public

~DCEObj - Destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The DCEObj class has associations with the following classes:
DOF (Aggregation)

4.3.3.4 DOF Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
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 incomming calls and redirect them to the appropriate implementation object, create/maintain Access Control Lists associated to the methods described in the service.

Attributes:

None

Operations:

None

Associations:

The DOF class has associations with the following classes:
None

4.3.3.5 ESO (CSS Provided-Class Inheriting from DCEServer Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
A generic Server class is provided at the server side. This class deals with the management of the objects that implement the interfaces. This class provides the functionality to interact with the objects that implement the interfaces. This class provides the functionality to interact with the naming and security services on behalf of the objects it manages. 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. The server main is a driver program which uses this class to create a deamon running all the time listening to incoming requests. Each time a request arrives, this application decides which server manager should address the request and spawns a new thread to run that implementaion of the server application. Each server driver can manage several server managers. A server manager is one implementaion of the server. This class provides all the needed interfaces to deal with core servies like the naming and the security. This class provides an abstraction of the interfaces to these core services, which the application programmer can use. The actual interface between this service and the core services is private and the application programmer never uses that directly. The number of requests a server main can take are limited, and can be set by the application programmer. It also maintains a queue to keep the incoming request if all the server implementaions are busy. The length of the queue is configurable by the application programmer. If a request arrives when the queue is full, then the request is ignored without any notifications. Application specific behavior can be extended by defining another object inheriting from the global server object (GSO). For example, if the application need to register it some local database, the register method can be implemented in the newly created global server object. This can be done by application programmer.

Attributes:

_Init - One time init flag
Data Type:int
Privilege:Protected
Default Value:

_activator - Activation object (optional)
Data Type:DCEActivation*
Privilege:Protected
Default Value:

_auth_arg_val - KeyTab file arg
Data Type:void*
Privilege:Protected
Default Value:

_auth_obj - Authorizer for management functions
Data Type:DCEMgmtAuthorizer*
Privilege:Protected
Default Value:

_authentication_service_type - Authn server type
Data Type:unsigned32
Privilege:Protected
Default Value:

_description - Text description
Data Type:char*
Privilege:Protected
Default Value:

_endpoints_registered - Flag, are the endpoints registered
Data Type:boolean
Privilege:Protected
Default Value:

_exports - List of Interfaces for export
Data Type:DCEInterfaceList
Privilege:Protected
Default Value:

_free_list - List of Interfaces for deletion
Data Type:DCEInterfaceList
Privilege:Protected
Default Value:

_group - Optional CDS group
Data Type:DCENsiGroup*
Privilege:Protected
Default Value:

_interface_registered - Flag, are interfaces registered with runtime
Data Type:boolean
Privilege:Protected
Default Value:

_interfaces - List of interfaces
Data Type:DCEInterfaceList
Privilege:Protected
Default Value:

_key_retrieval_obj - Key retriever object
Data Type:DCEKeyRetriever
Privilege:Protected
Default Value:

_max_call_requests - Maximum number of concurrent calls
Data Type:unsigned32
Privilege:Protected
Default Value:

_objects - List of server objects
Data Type:DCEObjectList
Privilege:Protected
Default Value:

_profile - Optional CDS profile
Data Type:DCENsiProfile *
Privilege:Protected
Default Value:

_profile_prio - Profile priority
Data Type:unsigned32
Privilege:Protected
Default Value:

_protocols - List of protocol sequences
Data Type:DCEProtocolList
Privilege:Protected
Default Value:

_protocols_registered - Flag, are protocols registered
Data Type:boolean
Privilege:Protected
Default Value:

_reaper - Cleanup thread
Data Type:DCEPthread*
Privilege:Protected
Default Value:

_servent - CDS server entry
Data Type:DCENsiEnry*
Privilege:Protected
Default Value:

_server_is_listening - Flag, is server object listen loop
Data Type:boolean
Privilege:Protected
Default Value:

_server_is_registered - Flag, is server registered in CDS
Data Type:boolean
Privilege:Protected
Default Value:

_server_lock - Main Mutex for DCEServer class
Data Type:pthread_mutex_t
Privilege:Protected
Default Value:

_server_principal_name - Principal name
Data Type:unsigned char*
Privilege:Protected
Default Value:

_use_protocols - Protocol registration policy
Data Type:DCEProtSeqReg
Privilege:Protected
Default Value:

Operations:

ExportObject - Exports the object information into the CDS.
Arguments:uuid_t*
Return Type:void
Privilege:Public

GetActivationObject - Used to access the Activation object associated with the server.
Arguments:
Return Type:DCEActivation*
Privilege:Public

GetAuthArg - The member functions defined above can be used to specify optional parameter information that will be passed to the key retrieval mechanism when keys are

requested. Security keys are stored in a key table, the default key table is a simple file with a well known filename. If the Server object is using the default key retrieval machanism but the key is not stored in the default key table the above functions may be used to set the filename of the key table to use. If a KeyRetriever object is registered with the server object, additional argument information can be passed to the KeyRetriever object by using the SetAuthArg function. The interpretation of the information set by the SetAuthArg member function is dependent on the value of the authentication service type.
Arguments:
Return Type:void*
Privilege:Public

GetAuthnService - Gets the authentication service type used to do the authentication between the Server object and the client.
Arguments:
Return Type:unsigned char*
Privilege:Public

GetDescription - Used to get a textual description for the server object.
Arguments:
Return Type:char*
Privilege:Public

GetEntry - Used to obtain the CDS class for the server objects entry name.
Arguments:
Return Type:DCENsiEntry*
Privilege:Public

GetExportScope - Gets the export policy that controls the registration of interface and object information with a Server object. By default, Interface and object information is registered with the local RPC runtime and the endpoint mapper.
Arguments:
Return Type:DCEExportScope
Privilege:Public

GetGroupName - Gets the group name of the entry from the CDS.
Arguments:
Return Type:const char*
Privilege:Public

GetHostPolicy - Gets the per host operation policy of the Server object. By default multiple instances of a particular server object can exist per host. This allows the option of enforcing a single instance per host.
Arguments:
Return Type:DCEHostPolicy
Privilege:Public

GetKeyRetriever - Gets the key retriever object associated with the server. This is used when an application programmer wants to run a server with an identity different from the principal starting that server.
Arguments:
Return Type:DCEKeyRetriever*
Privilege:Public

GetManagementAuthorizer - Gets a MgmtAuthorizer object for a Server object. If the Server Object has no MgmtAuthorizer object, then the default DCE policy is used to control access to the server management functions. However, if such an object is registered it will be used to control access to management functionality.
Arguments:
Return Type:DCEMgmtAuthorizer*
Privilege:Public

GetName - Gets the server entry name from the CDS. Cell relative or a global names can be used to denote a server entry.
Arguments:
Return Type:char*
Privilege:Public

GetPrincipalName - Gets the security principal's name of the Server object.
Arguments:
Return Type:unsigned char*
Privilege:Public

GetProfile - Returns a CDS profile name.
Arguments:
Return Type:DCENsiProfile*
Privilege:Public

GetProfileName - Retrieves profile information from the CDS.
Arguments:
Return Type:char*
Privilege:Public

GetProfilePriority - Used to retrieve profile priority information from the CDS.
Arguments:
Return Type:unsigned32
Privilege:Public

Listen - Keeps the server in the listen state waiting for incoming requests. Maximum number of concurrent calls can be optionally specified here.
Arguments:

Return Type:void
Privilege:Public

RegisterObject - Registers a new object with the server and optionally with the CDS. The server must have at least one object registered before the Listen method can be called.
Arguments:
Return Type:void
Privilege:Public

RemoveObjects - Server manager objects that are unregistered from the server object may be placed on an internal list. The RemoveObjects function can be used to free up resources associated with the manager objects. This function is called automatically by a garbage collection thread.
Arguments:
Return Type:void
Privilege:Public

ServerCleanup - Implements the cleanup handler for the server when the server goes down like removing binding information from the CDS.
Arguments:void*
Return Type:static pthread_addr_t
Privilege:Public

ServerIsListening - Returns true if the Server object is currently in the listening state and false otherwise.
Arguments:
Return Type:boolean
Privilege:Public

SetActivationObject - SetActivationObject is used to register an Activation object with the server. The activation object can be associated with a single interface or a vector of interfaces.
Arguments:DCEActivation*,DCEUuidVector&
Return Type:void
Privilege:Public

SetActivationObject - SetActivationObject is used to register an Activation object with the server. The activation object can be associated with a single interface or a vector of interfaces.
Arguments:DCEActivation*,DCEUuid&
Return Type:void
Privilege:Public

SetAuthArg - The member functions defined above can be used to specify optional parameter information that will be passed to the key retrieval mechanism when keys are

requested. Security keys are stored in a key table, the default key table is a simple file with a well known filename. If the Server object is using the default key retrieval machanism but the key is not stored in the default key table the above functions may be used to set the filename of the key table to use. If a KeyRetriever object is registered with the server object, additional argument information can be passed to the KeyRetriever object by using the SetAuthArg function. The interpretation of the information set by the SetAuthArg member function is dependent on the value of the authentication service type.
Arguments:arg:void*
Return Type:void*
Privilege:Public

SetAuthInfo - Registers the principal name and authorization (optionally a key retriever) information with the runtime RPC. Two versions of the SetAuthInfo member function are provided. The first one allows a KeyRetriever object to be set for the Server object allowing an alternative mechanism for retrieving security keys to be used. The second version, just allows the setting of principal name and authentication service type. In this case the default key retrieval mechanism provided by DCE is used.
Arguments:unsigned char*, unsigned32, DCEKeyRetriever*, void*
Return Type:void
Privilege:Public

SetAuthInfo - Registers the principal name and authorization (optionally a key retriever) information with the runtime RPC. Two versions of the SetAuthInfo member function are provided. The first one allows a KeyRetriever object to be set for the Server object allowing an alternative mechanism for retrieving security keys to be used. The second version, just allows the setting of principal name and authentication service type. In this case the default key retrieval mechanism provided by DCE is used.
Arguments:unsigned char *, unsigned32, void*
Return Type:void
Privilege:Public

SetAuthnService - Sets the authentication service type used to do the authentication between the Server object and the client.
Arguments:this_service:unsigned32
Return Type:void
Privilege:Public

SetDescription - Used to obtain the name syntax of the server if specified.
Arguments:desc:char*
Return Type:void
Privilege:Public

SetExportScope - Sets the export policy that controls the registration of interface and object information with a Server object. By default, Interface and object information is registered with the local RPC runtime and the endpoint mapper.

Arguments:scope:DCEExportScope
Return Type:void
Privilege:Public

SetGroupName - Sets the group name of the entry from the CDS.
Arguments:cons DCENsiName&
Return Type:void
Privilege:Public

SetGroupName - Sets the group name of the entry from the CDS.
Arguments:const unsigned char*, const unsigned32
Return Type:void
Privilege:Public

SetGroupName - Sets the group name of the entry from the CDS.
Arguments:const char*,const unsigned32
Return Type:void
Privilege:Public

SetHostPolicy - Sets the per host operation policy of the Server object. By default multiple instances of a particular server object can exist per host. This allows the option of enforcing a single instance per host.
Arguments:policy:const DCEHostPolicy
Return Type:void
Privilege:Public

SetKeyRetriever - Sets the key retriever object associated with the server. This is used when an application programmer wants to run a server with an identity different from the principal starting that server.
Arguments:kr:DCEKeyRetriever*
Return Type:void
Privilege:Public

SetManagementAuthorizer - Sets a MgmtAuthorizer object for a Server object. If the Server Object has no MgmtAuthorizer object, then the default DCE policy is used to control access to the server management functions. However, if such an object is registered it will be used to control access to management functionality.
Arguments:ma:DCEMgmtAuthorizer*
Return Type:void
Privilege:Public

SetName - Sets the server entry name from the CDS. Cell relative or a global name can be used to denote a server entry.
Arguments:const char*,const unsigned32
Return Type:void

Privilege:Public

SetName - Sets the server entry name from the CDS. Cell relative or a global name can be used to denote a server entry.
Arguments:const unsigned char*,const unsigned32
Return Type:void
Privilege:Public

SetPrincipalName - Sets the security principal's name of the Server object.
Arguments:unsigned char*
Return Type:void
Privilege:Public

SetProfileName - Registers a profile entry in the CDS.
Arguments:const char*, const unsigned32
Return Type:void
Privilege:Public

SetProfileName - Registers a profile entry in the CDS.
Arguments:const unsigned char*,const unsigned32,
Return Type:void
Privilege:Public

SetProfileName - Registers a profile entry in the CDS.
Arguments:const DCENsiName&
Return Type:void
Privilege:Public

SetProfilePriority - Used to manipulate the CDS profile information. Specifically, used to set profile priority information, or to associate a priority to the entry.
Arguments:prio:unsigned32
Return Type:void
Privilege:Public

Shutdown - Stops the server for new requests, unregisters the information from the CDS.
Arguments:
Return Type:void
Privilege:Public

Stop - Causes the server to stop listening for new incoming requests.
Arguments:
Return Type:void
Privilege:Public

UnExportInterface - Unexports the server interface information from the CDS.

Arguments:DCEInterfaceMgr&
Return Type:void
Privilege:Public

UnExportObject - Removes the object information from the CDS.
Arguments:uuid_t*
Return Type:void
Privilege:Public

UnRegisterObject - Unregisters a new object with the server and optionally with the CDS. The server must have at least one object registered before the Listen method can be called.
Arguments:DCEInterfaceMgr&, boolean32
Return Type:void
Privilege:Public

UnRegisterObject - Unregisters a new object with the server and optionally with the CDS. The server must have at least one object registered before the Listen method can be called.
Arguments:uuid_t*, boolean32
Return Type:void
Privilege:Public

UnRegisterObject - Unregisters a new object with the server and optionally with the CDS. The server must have at least one object registered before the Listen method can be called.
Arguments:DCEObj&, boolean32
Return Type:void
Privilege:Public

UseAllProtocols - Uses all the protocols and sets the maximum number of request that can be handled by the server object.
Arguments:const unsigned32
Return Type:void
Privilege:Public

UseAllProtocols - Uses all the protocols and sets the maximum number of request that can be handled by the server object.
Arguments:DCEInterfaceMgr&, const unsgined32
Return Type:void
Privilege:Public

UseProtocol - Instructs the server to use a particular protocol for client requests. This can be called multiple times to set multiple protocols.
Arguments:const unsigned char*,const unsigned32
Return Type:void
Privilege:Public

UseProtocol - Instructs the server to use a particular protocol for client requests. This can be called multiple times to set multiple protocols.
Arguments:const unsigned char*,const unsigned char*,const unsigned32
Return Type:void
Privilege:Public

UseProtocol - Instructs the server to use a particular protocol for client requests. This can be called multiple times to set multiple protocols.
Arguments:const unsigned char*,DCEInterfaceMgr&,const unsigned32
Return Type:void
Privilege:Public

_AuthFunc - Protected member function.
Arguments:handle_t,unsigned32,unsigned32*
Return Type:boolean32
Privilege:Public

_GetInterfaces - Returns a pointer to a list of each unique interface provided by the objects.
Arguments:
Return Type:DCEInterfaceList*
Privilege:Public

_GetKey - Protected member function.
Arguments:void*,unsigned_char_p_t,unsigned32,void**,unsigned32*
Return Type:void
Privilege:Public

_GetObjectList - Returns a pointer to a list of Object_Set_t structures defined in the Server.H file which contains object and export information for every object registered with the server object.
Arguments:
Return Type:DCEObjectList*
Privilege:Public

_RegisterObject - Register the server object passing an instance of the DCEInterfaceMgr class.
Arguments:DCEInterfaceMgr&,const boolean32
Return Type:void
Privilege:Public

Associations:

The ESO class has associations with the following classes:
DOF (Aggregation)

4.3.3.6 Naming Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Implementation of the Directory Naming Service.

Attributes:

None

Operations:

None

Associations:

The Naming class has associations with the following classes:
DOF (Aggregation)

4.3.3.7 Security Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Implementation of the Security Service.

Attributes:

None

Operations:

None

Associations:

The Security class has associations with the following classes:
DOF (Aggregation)

4.3.4 Functional Model

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, 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.

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 users. 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.

Table 4.3.4-1 describes the DOF classes.

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.

Table 4.3.4-1. 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.

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.

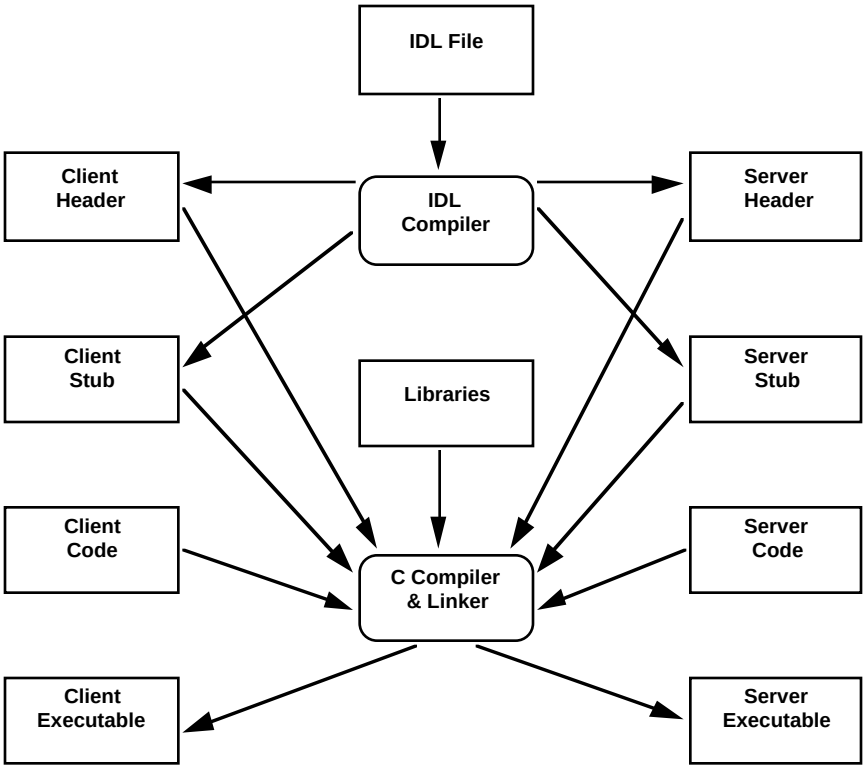

Figure 4.3.4-1. Client/Server Application Development Process

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 Dynamic Model Scenarios

4.3.5.1 Scenario #1

4.3.5.1.1 Abstract

This scenario traces the interactions of a client application, a Dataserver search client object, that takes some search criteria and sends it to an application server, a Dataserver search server object, running remotely that can perform the search request and return the results.

4.3.5.1.1 Interfaces

SDPS client application.

4.3.5.1.2 Stimulus

Users want to search a database given some search criteria and invokes the search method.

4.3.5.1.3 Desired Response

Results of the search should be returned to the client that then can be used to display a picture.

4.3.5.1.4 Participating Classes

ESO

DCEInterface

DCEInterfaceMgr

DCEObj

4.3.5.1.5 Pre-conditions

The SDPS Search interface is defined in the IDL file and is compiled using OODCE idl++ compiler, which will generate an abstract class definition from the IDL implementation and a

concrete class definition that is derived from the abstract class on both the client and the server. The search client and the search server are developed from these classes and compiled. The search application server is running on a remote host. Its location is entered in the namespace so clients can find it by a name. The search method is protected by Access Control Lists so only authorized clients can invoke this method.

4.3.5.1.6 Post-conditions

The results of the search are returned to the application client.

4.3.5.1.7 Scenario Description

The client application creates a proxy object of the search server with the given search server name. The constructor (inherited from the interface class) consults the central namespace to find the location of the server (unique name given to the server) and binds to it. The proxy object is returned to the client application. The client application then invokes an operation on the proxy object. The proxy object in turn forwards that call to the server manager. The server manager application first gets the client's privilege certificate and consults the security server to authenticate the client. The server manager then invokes the appropriate server implementation method passing the arguments. The actual implementation method gets the client's privileges and consults the security server (ACL Manager calls) to check whether the client is authorized to invoke the method. The server code executes the search request, and returns the results back to the client.

4.3.5.1.8 Event Trace

4.3.5.2 Scenario #2

4.3.5.2.1 Abstract

A server object is running and is registered in the namespace. When the client object is constructed, location information can be supplied to the constructor via arguments. By default no location information is required, in which case any manager object that implements the required interface will be found and used. In this scenario when the search client is constructed, location information is supplied in the constructor using the CDS name. This name can refer to a server, a group, or a profile.

4.3.5.2.2 Interfaces

SDPS client application.

4.3.5.2.3 Stimulus

Create a proxy object and provide a CDS Name to locate manager objects. The name can be a server, a group or a profile.

4.3.5.2.4 Desired Response

Find the desired manager object.

4.3.5.2.5 Participating Classes

ESO

DCEInterface

DCEInterfaceMgr

DCEObj

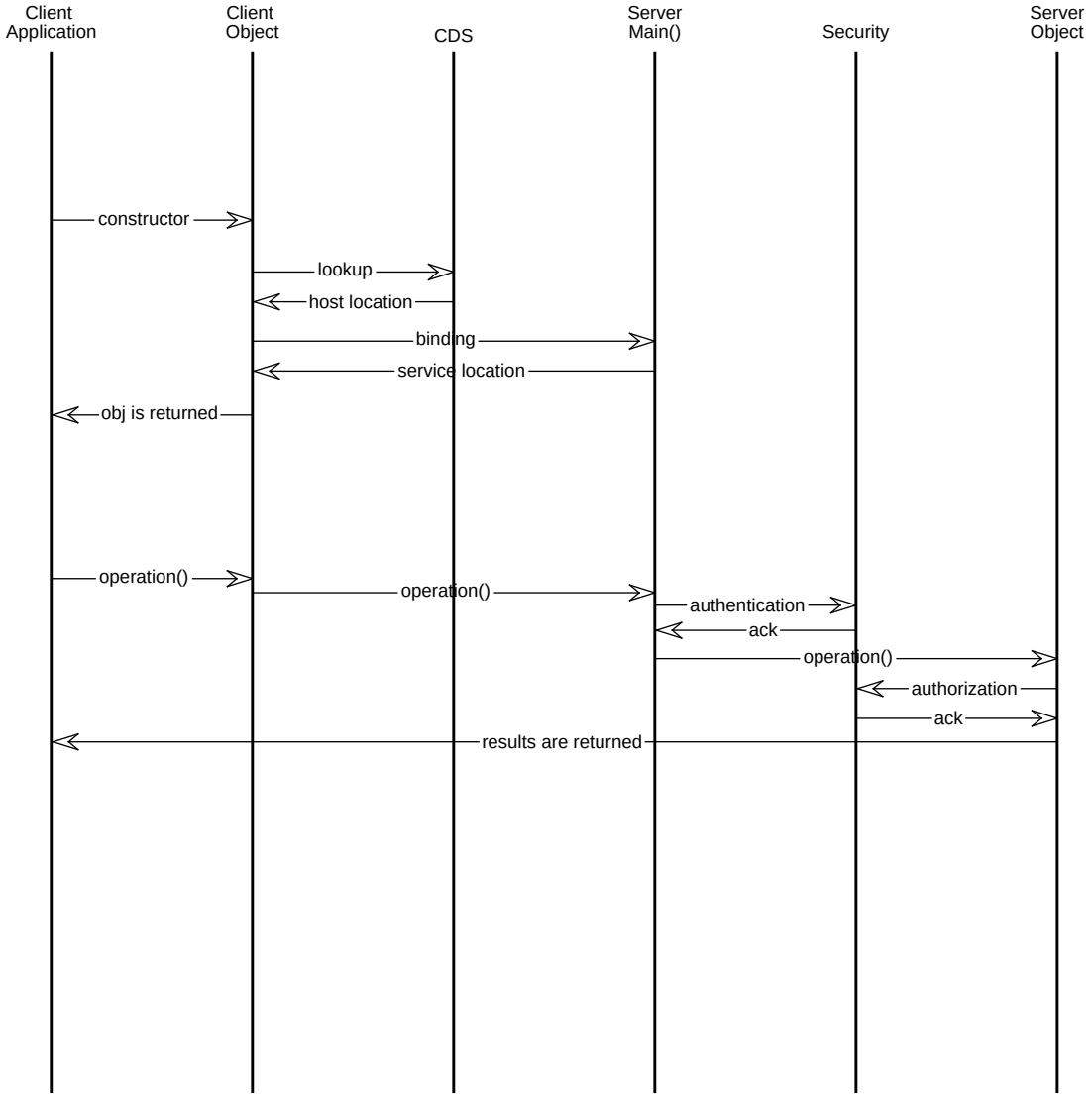

Figure 4.3.5-1. Distributed Object Framework Event Trace

4.3.5.2.6 Pre-conditions

CDS up and running

4.3.5.2.7 Post-conditions

The client object locates the manager object given the CDS name.

4.3.5.2.8 Scenario Description

A server object is running and is registered in the namespace. A caller creates a proxy object supplying a logical name (CDS name). The client runtime consults the namespace, obtains partial binding information. Consults the endpoint mapper and obtains the port number. Gets all the binding information and bind to the server. Now any calls made to the proxy object are routed to the real server object.

4.3.5.2.9 Event Trace

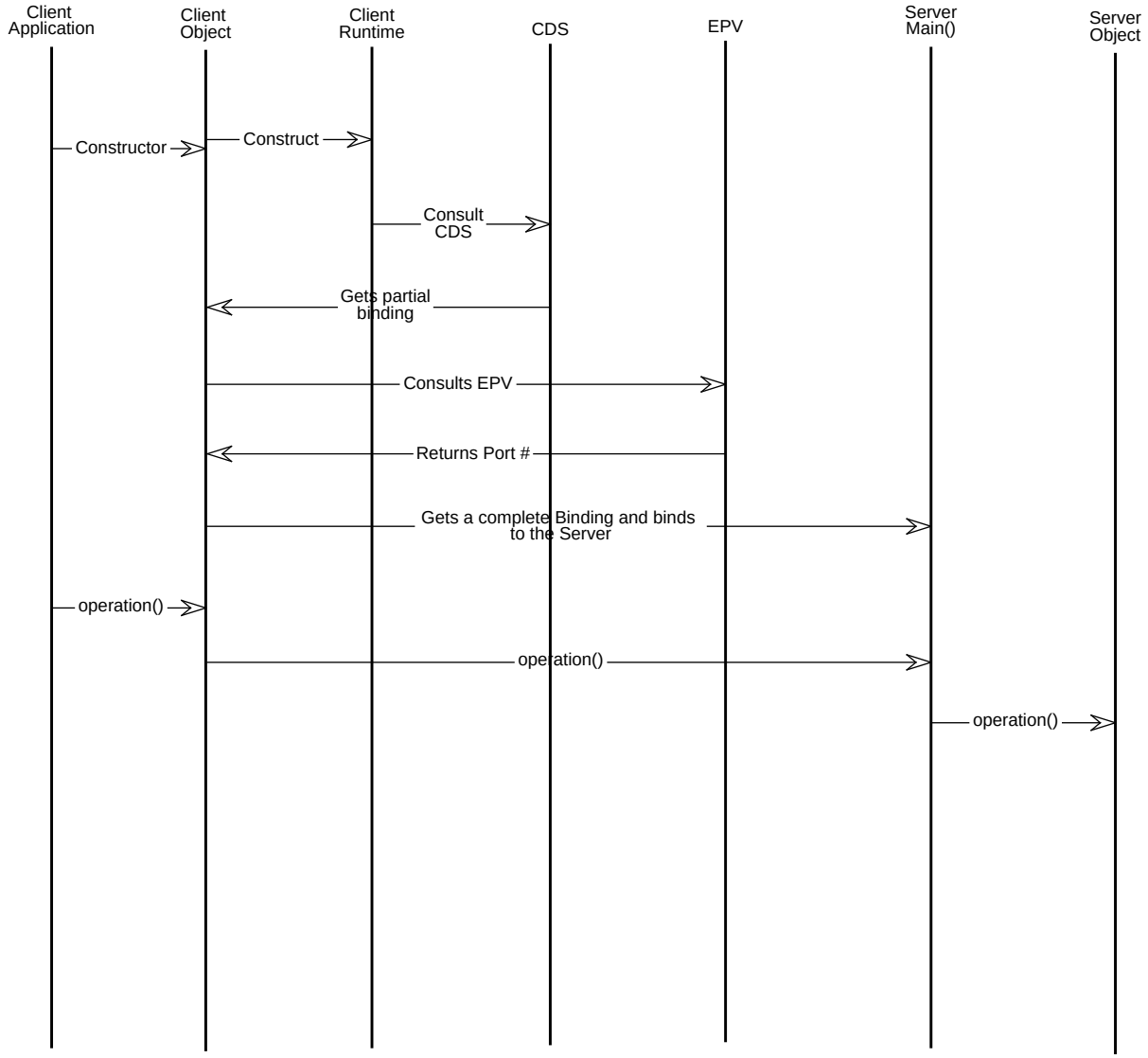

Figure 4.3.5-2. Distributed Object Framework Event Trace

4.3.5.3 Scenario #3

4.3.5.3.1 Abstract

A proxy object is created with no arguments. When the SDPS search client object is constructed, location information is supplied to the constructor using the interface as an implicit argument.

4.3.5.3.2 Interfaces

SDPS client application.

4.3.5.3.3 Stimulus

Create a proxy object and provide the interface to locate SDPS server manager objects.

4.3.5.3.4 Desired Response

Find the desired manager object.

4.3.5.3.5 Participating Classes

ESO

DCEInterface

DCEInterfaceMgr

DCEObj

4.3.5.3.6 Pre-conditions

CDS up and running.

4.3.5.3.7 Post-conditions

Any object that implements the requested interface will be bound.

4.3.5.3.8 Scenario Description

Client runtime knows the interface id of the distributed object. It consults the namespace and finds the first SDPS search server object whose interface matches the interface id that it is looking for and gets the binding information (from the port map also). The search client proxy binds to the server object.

Calls made to the proxy are routed to the search server object and results are returned back to the search client application through the proxy.

4.3.5.3.9 Event Trace

4.3.5.4 Scenario #4

4.3.5.4.1 Abstract

There can be any number of failures while creating and invoking methods on a remote application, such as, network and communication errors. The application, either the client or the server portion of it, need to be able to detect and process such failures.

The OODCE library uses C++ exceptions as the model for error handling. This provides a clean and consistent model for dealing with problems that can occur when using DCE. The caller of the OODCE library should be prepare to handle exceptions that are raised within the library. All OODCE calls including constructors should be surrounded by a C++ **try/catch** clause.

4.3.5.4.2 DCEInterface

SDPS client application.

4.3.5.4.3 Stimulus

An error occurred within the client (proxy) code.

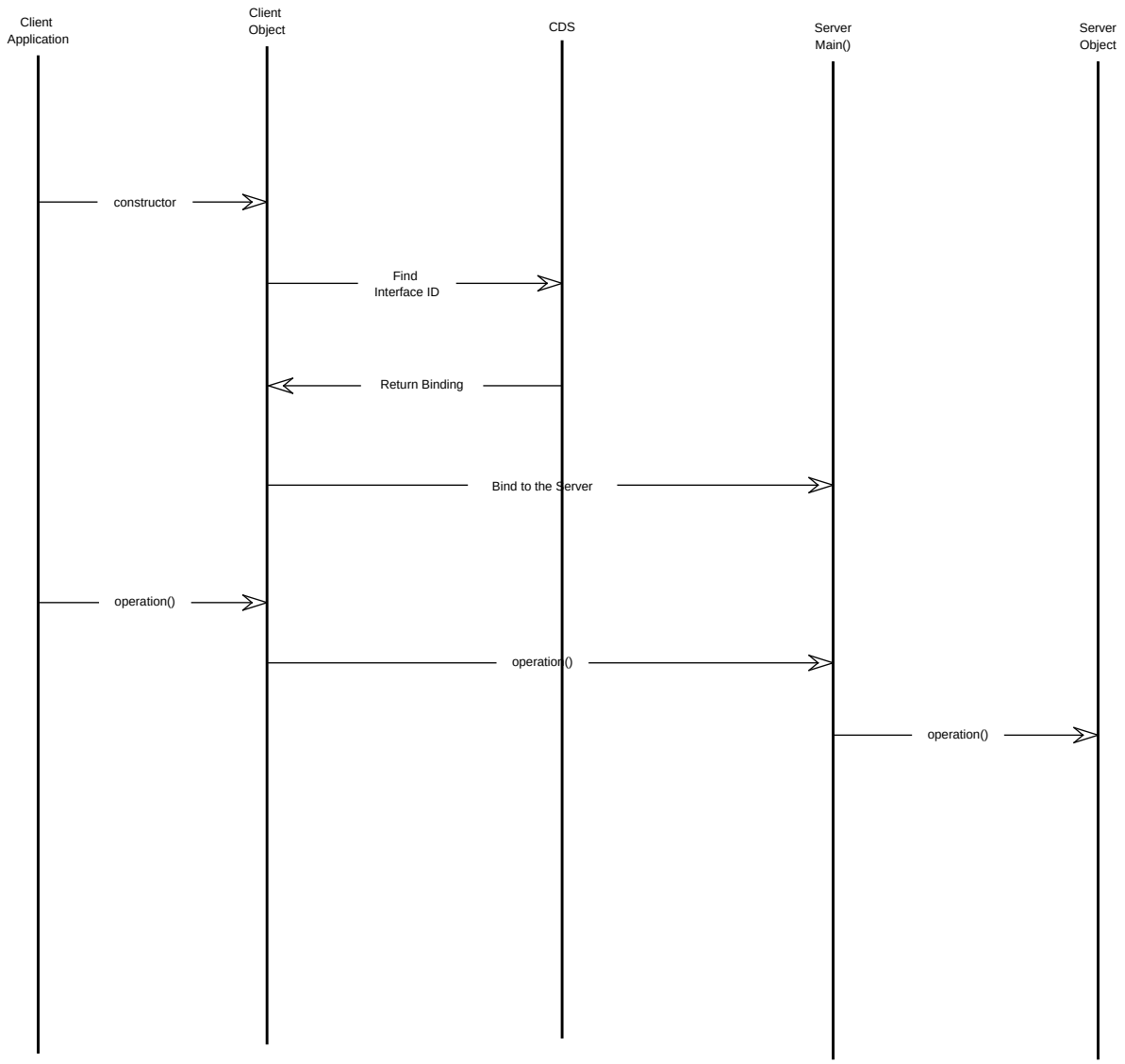

Figure 4.3.5-3. Distributed Object Framework Event Trace

4.3.5.4.4 Desired Response

A fatal condition is trapped and transmitted to the user.

4.3.5.4.5 Participating Classes

ESO

DCEInterface

DCEInterfaceMgr

DCEObj

4.3.5.4.6 Pre-conditions

Setup a try block to catch exceptions prior to constructing a client object and calling the member function that invokes the remote procedure on the server side object.

4.3.5.4.7 Post-conditions

Exceptions are caught.

4.3.5.4.8 Scenario Description

This scenario shows the use of the try/catch clause. The creation of the proxy and the invocation calls are encapsulated in a C++ try block with a corresponding catch clause. Any communication/ status errors result in C++ exceptions and is caught by the application programmer to act appropriately.

The catch block traps any DCE errors that may occur during the execution of the server main function. An exception type is converted into a character string and a user message is printed. In general, every public status code defined by DCE has a corresponding exception class in OODCE. Each of these classes can be converted back into a status code (error_status_t) or to a character string that describes the error and the DCE subsystem that generated it. Since specific exception classes exist for each DCE status code, specific DCE fault conditions can be trapped and processed.

4.3.5.4.9 Event Trace

4.3.5.5 Scenario #5

4.3.5.5.1 Abstract

Creating a factory manager object to allow for the creation and deletion of other manager objects (sessions objects) provided by the server.

Objects that are declared before the call to server listen are static and are available when the server program executes. OODCE allows objects to be created or activated dynamically (that is while the server program is listening for client requests) on behalf of a client. This scenario describes creating a factory object that will be able to create proxy objects, and the next scenario (#6) describes object activation.

The application programmer using OODCE is able to create manager objects within a server process. Creation is different than activation in that assumes that an object has already been created. OODCE can support a factory by specifying an IDL interface for a factory object. The factory manager object on the server is then responsible for creating other manager objects provided by the server. For example, consider the case of the dataserver interface. The class DsDdRequesManagerS represents a factory server object, and its job is to create DsDdDistRequestS manager objects.

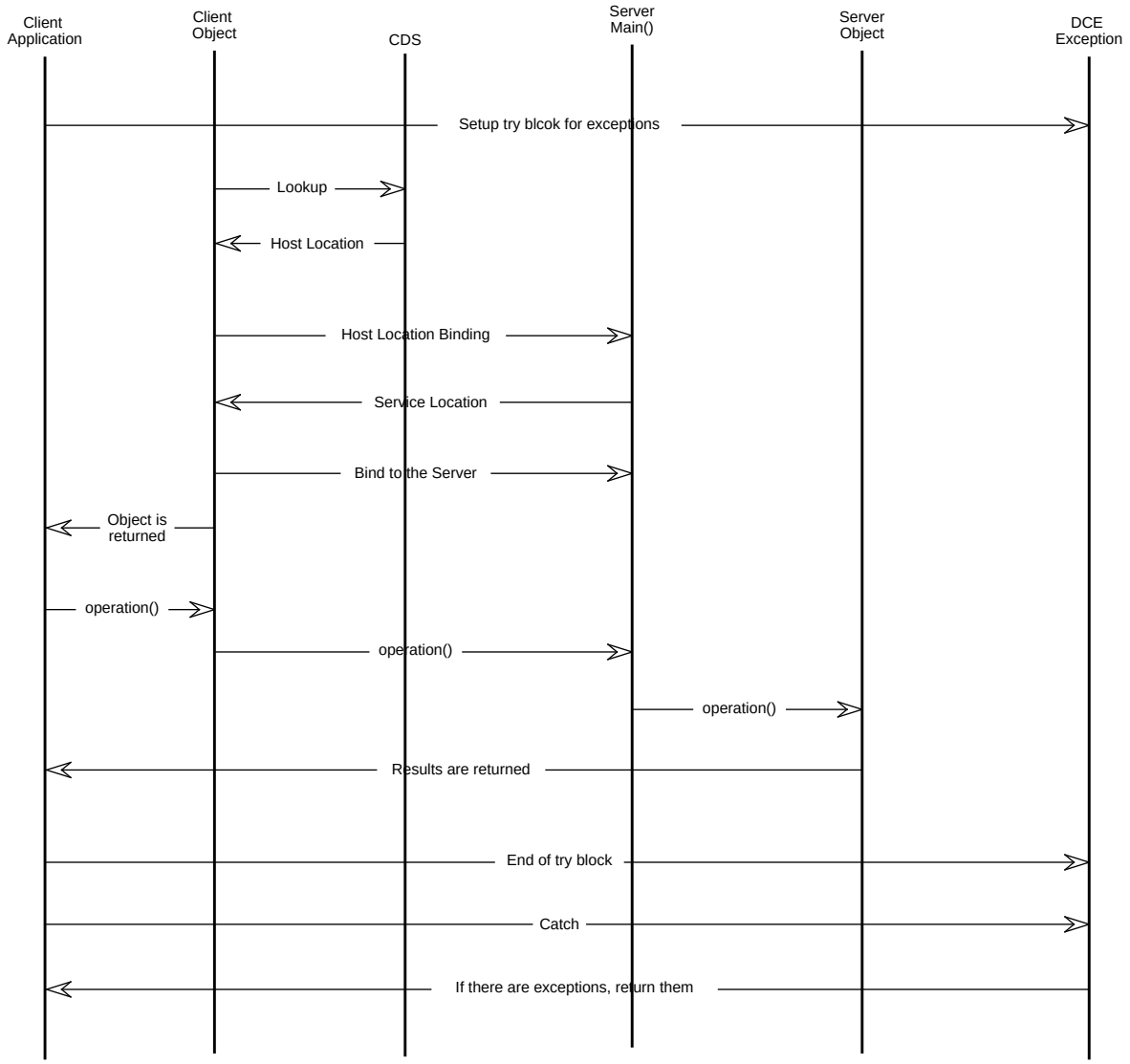

Figure 4.3.5-4. Distributed Object Framework Event Trace

To allow for factory objects the support of different object types, the programmer can assign the application interface UUID to the factory object UUID. The interface UUID will be common to all the newly created objects, but each object type will be identified by a different object UUID. For example, take the case of the dataserver, this object will have an interface identifier, but there is one dataserver per DAAC. Each one of them will have the same interface UUID but a unique object UUID. When the user tries to connect to the dataserver in Alaska, it needs to supply the object UUID in order to be able to locate it (the interface id will be implicitly supplied at the construction time.

Factories can also be used to store information related to each manager object that was created.

4.3.5.5.2 Interfaces

SDPS client application.

4.3.5.5.3 Stimulus

An object needs to get created.

4.3.5.5.4 Desired Response

A factory object gets created and is registered with the Global Server Object (GSO).

4.3.5.5.5 Participating Classes

ESO

DCEInterface

DCEInterfaceMgr

DCEObj

4.3.5.5.6 Pre-conditions

The object is not declared before the call to server listen (so it needs to be created dynamically).

4.3.5.5.7 Post-conditions

A factory object gets created and is registered with the GSO.

4.3.5.5.8 Scenario Description

This scenario shows how a factory object on the server creates and deletes other manager objects to be managed by the server. The interface definition language file for the factory provides RPC specifications (methods) for creating and deleting objects at the server side using object references.

The factory server is brought up, that is, an instance of the factory manager object, 'Factory_1_0_Mgr' (a class generated by the IDL compiler) is created, registered with the GSO and set to listen. Its job is to create and delete proxy objects.

By means of the factory manager an application client can create other manager objects (within the server program) when the application client object is constructed. The client application constructs a factory client object which creates an instance of the factory object to allow for creation of application objects. Using this newly created factory object, the factory client invokes the method 'Create' at the server side. A new *application* manager object is created by the factory manager object, which inherits from the IDL generated class '*application*_1_0_Mgr', and gets registered with the GSO (which in turns, registers it with the runtime to take incoming calls). The factory manager object returns back a reference to the created object. This object reference is used by the factory client object to set the binding handle, and to initialize the binding information of the client object to refer to the application manager object just created by the factory. The client application is then able to make remote calls.

The application manager object gets deleted when the client object's destructor is run.

4.3.5.5.9 Event Trace

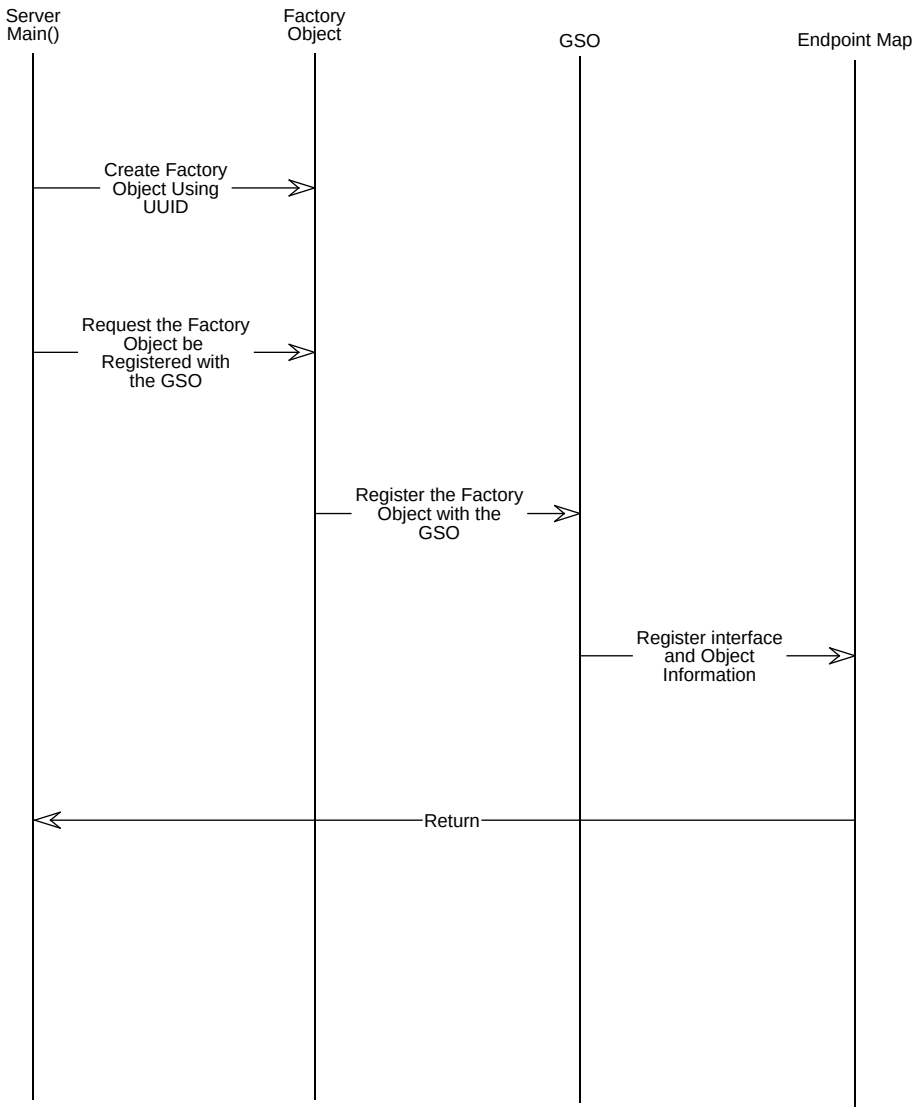

Figure 4.3.5-5. Distributed Object Framework Event Trace

4.3.5.6 Scenario #6

4.3.5.6.1 Abstract

Creating an activation object, attaching it to the server object, and registering it with the GSO.

If a server program manages a large number of DCE objects, it can be impractical to have all of its objects active at the same time. Activation is different from creation because it assumes that the object has already been created and initialized but is not currently active or available for clients to use (it is in a passive state). Object activation allows a server designer to choose when it is appropriate to activate an object (normally when a client needs to use it) and do this independent of when a server program is started.

Activation is generally associated with persistent storage. When an object is passive, it usually means that its state has been saved in long term storage (i.e.: a file system or database). OODCE does not provide persistent storage of an object's state, but can work with persistent store implementations like object databases to achieve the semantics of object activation. The application programmer needs to provide the read/write methods of the object's state to the UNIX file system.

For an object to be activated, the DCE object UUID is required to identify the object to be activated. OODCE defines an abstract class called *Activation* that provides an abstraction onto the activation of manager objects. You need to derive a concrete class from the Activation class and provide an implementation that performs the activation for a specific object class or set of classes. This implementation locates the object state based on the DCE object UUID, creates a new manager object for the class associated with the Object UUID, initializes the new manager object with the state and registers it with the GSO.

4.3.5.6.2 Interfaces

SDPS client application.

4.3.5.6.3 Stimulus

An object needs to get activated.

4.3.5.6.4 Desired Response

Object is activated.

4.3.5.6.5 Participating Classes

ESO

DCEInterface

DCEInterfaceMgr

DCEObj

4.3.5.6.6 Pre-conditions

The object has already been created and initialized but it is not currently active or available for clients to use. Its state information was saved on disk.

4.3.5.6.7 Post-conditions

Object is activated, its state information is recovered from the disk.

4.3.5.6.8 Scenario Description

This scenario will demonstrate how to locate the object state based on the UUID, create a new manager object for the class associated with the object UUID, initialize the new manager object with the state, and register the manager object with the GSO.

For an application to support Activation, a server program must register an activation object implementation with the GSO. This is done in the server program's main function. It sets an activation object by passing the application interface UUID. Then the server is set to listen. When

the server main code executes, it creates an application server with a single manager object. An activation object is registered with the GSO using the SetActivationObject member function. This function is called with the activation object and an interface UUID. Activation is done when a client makes a call on an interface. The GSO needs to know which interfaces support activation. In this scenario activation only occurs when a client object makes a call to the server program.

SetActivationObject can be called more than once for different interfaces or a vector of interface UUIDs can be passed to the member function instead of a single UUID.

Once the server program has been started and the initial manager object has been registered in the endpoint map, any application client program can activate that manager object if necessary.

To implement Activation, the manager object needs to define a class that is derived from the base DCEActivation abstract class. This new class must be implemented to support activation of application objects. In general a single member function needs to be implemented called ActivateObject. 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. This UUID is used to check if the object was previously created, if so, its state is read from the file, otherwise, the state is set to zero. The constructor initializes the object state from the file system. Then registers the application manager object with the GSO and returns back the activation result. Before the constructor exits, it creates a thread whose job will be to periodically check the object states every certain period. If there have been no client requests on the manager object between two checkpoints, its state (any type of information, i.e. a math computation) is stored in a UNIX file system and the manager object is unregistered from the server program. If a client request is made on this manager object, the object is reactivated from its persistent storage (store state of the file system) for further bookkeeping/maintenance.

4.3.5.6.9 Event Trace

See Figure 4.3.5-6.

4.3.6 Implementation

COTS products are available that implement this architecture. Some of them are not mature, and some of them do not provide the full object functionality. It has been recommended that an encapsulated DCE (using OODCE) be used now which provides security, object oriented capabilities, and a high degree of interoperability, and migrate to a CORBA product in Release C. Encapsulated DCE will provide all the functionality described in DOF.

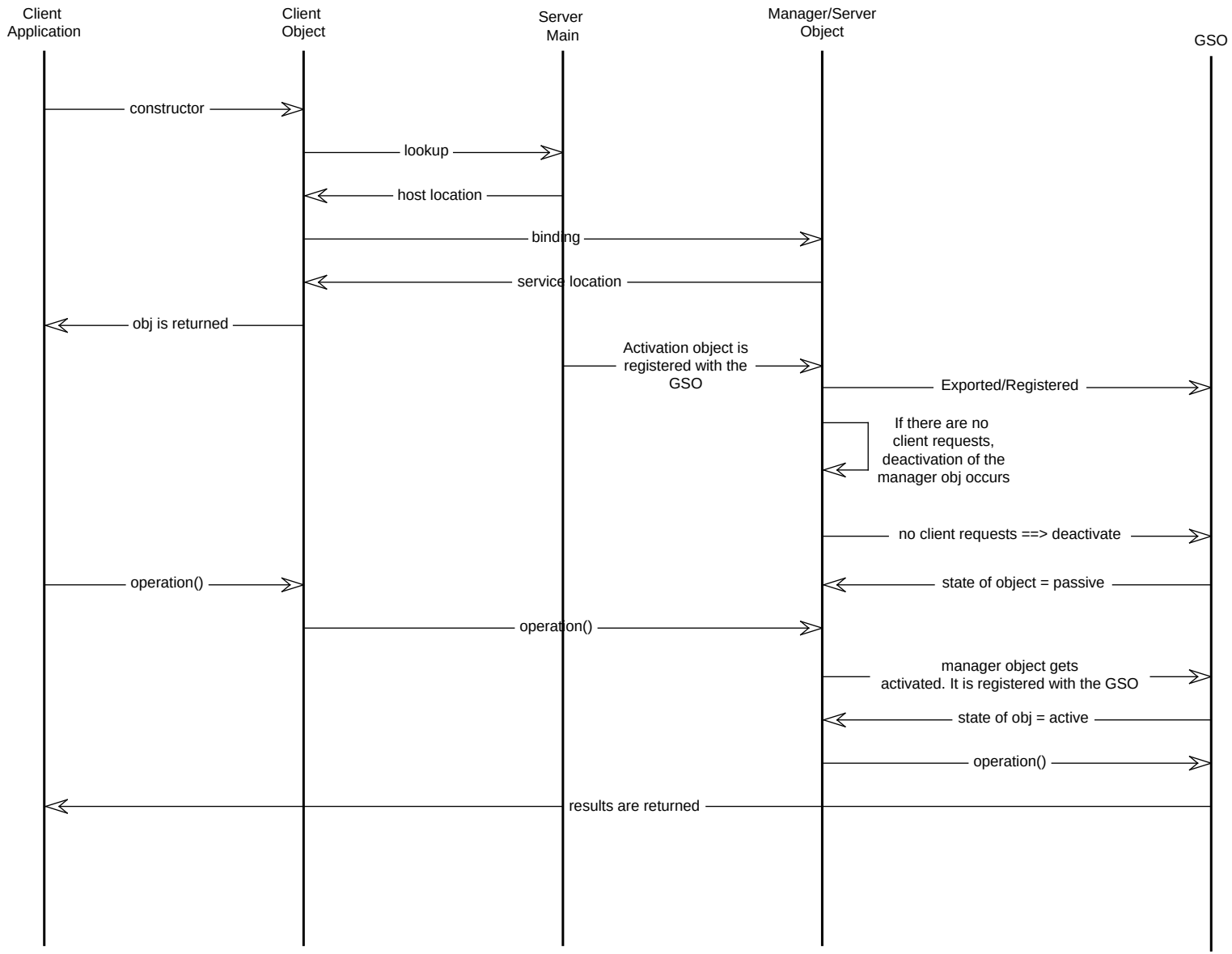

Figure 4.3.5-6. Distributed Object Framework Event Trace

4.3.7 Service/CSCI Management and Operation

4.3.7.1 System Management and Strategy

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. Please refer to each individual service description for more details.

4.3.7.2 Operator Interface

Not applicable.

4.3.7.3 Reports

None.

4.3.8 Frequently Asked Questions

Some of the Frequently Asked Questions (FAQ) included in this section are taken from the OSF DCE FAQ. We have included those that are directly relevant to ECS. Refer to the OSF Web Page (http://www.osf.org/dce/faq-mauney.html) for the most up to date and complete version of the FAQ.

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, going back to the Dataserver 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 consists 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.

Refer to the MSS documentation for details on how Mode Management will be implemented by ECS applications using some of the ideas presented here.

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.

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 similiar to other RPC implementations when doing the same things. The throughput and response times for a series of remote procedure calls is similiar.

The use of features in DCE not present in other implementations may consume additional time and resources. For example, name-based binding may required 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.

IBM has done quite a bit of performance testing of DCE. Many of the reports are available on line; go to IBM's corporate Web page, http://www.ibm.com, and choose the search feature to search for something like "dce performance". To summarize a few of the results in these reports:

RPC time increases linearly with the size of the data.

Passing a large array in one RPC call is about the same speed as using a pipe

assing a large array in several RPC calls is slower than a single call using a pipe

The "packet integrity" security level slows the RPC, nearly doubling the total time for calls.

The "packet privacy" level incurs a several-fold increase in time.

RPC is slower than simply making socket calls directly.

The time spent on RPC overhead was a small fraction of the total

processing time in a realistic business scenario.

RPC is pretty fast on Intel procesors (running OS/2).

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 then 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 IDLs 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.

DCE V1.1 supports the GSSAPI, so non-DCE services that use GSSAPI can be integrated with DCE security server.

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 you 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 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 created 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 a 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 scalable 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 interactive (for operators) and development (API, for applications) interfaces to manage electronic mail messages. An operator, for this discussion is defined as a member of the M&O staff. It is not the intention of the ECS to provide interactive mail services or mail stores for external end users. Interactive interfaces provide a full set of functionality, and are entirely COTS. The API is limited in scope as it only addresses sending a message from within an application ("mail-enabled application") There is no identified requirement to provide a secure Email service.

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 interface encapsulates the interface to OTS mail (SMTP). There are mail browsers and clients distributed with each vendor OS available for users.

4.4.1.3 Object Model

See Figure 4.4.1.3-1.

4.4.1.3.1 CsBBMailRelA Class

Parent Class:CsEmMailRelA
Public:Yes
Distributed Object:No
Purpose and Description:
Used to post messages to bulletin boards.

Attributes:

NNTPHost - Will hold the name of the NNTP host to send the message to.
Data Type:EcTChar[EcDShortStr]
Privilege:Private
Default Value:

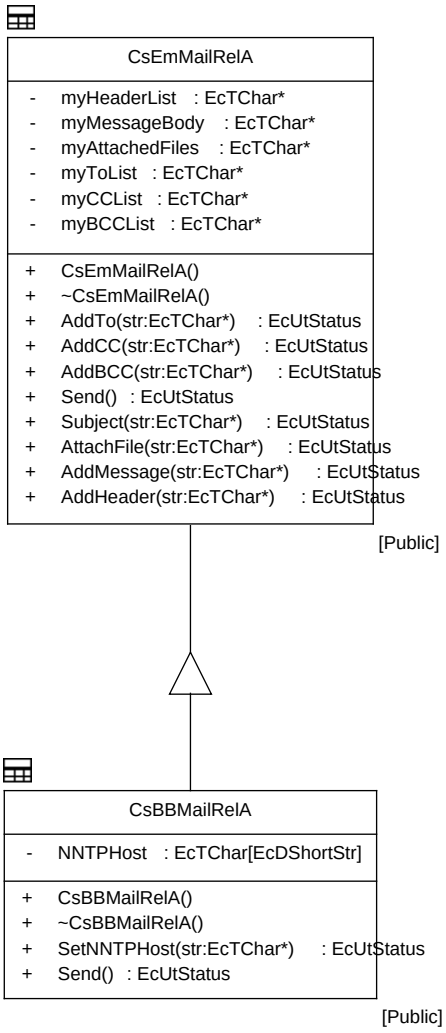

Figure 4.4.1.3-1. Email Service Object Model

Operations:

CsBBMailRelA - Constructor that will create object and send the specified message. It will not delete any storage after the message has been sent.
Arguments:
Return Type:Void
Privilege:Public

Send - Will send the message in the current state. It will not delete any of the internally stored data.
Arguments:
Return Type:EcUtStatus
Privilege:Public

SetNNTPHost - Will set the NNTP host to post the message to.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

~CsBBMailRelA - Destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The CsBBMailRelA class has associations with the following classes:
None

4.4.1.3.2 CsEmMailRelA Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
Used to send email to recipients.

Attributes:

myAttachedFiles - Internal stored (possibly uuencoded) message bodies.
Data Type:EcTChar*
Privilege:Private
Default Value:

myBCCList - Internal list of BCC recipients of the message.
Data Type:EcTChar*
Privilege:Private
Default Value:

myCCList - Internal list of CC's recipients of the message.
Data Type:EcTChar*
Privilege:Private
Default Value:

myHeaderList - Internal list of headers.
Data Type:EcTChar*
Privilege:Private
Default Value:

myMessageBody - Internal copy of the message body.
Data Type:EcTChar*
Privilege:Private
Default Value:

myToList - Internal list of recipients of the message.
Data Type:EcTChar*
Privilege:Private
Default Value:

Operations:

AddBCC - Will add the name string to the list of BCC'd recipients. Commas may be used to separate multiple usernames.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

AddCC - Will add the name string to the list of CC'd recipients. Commas may be used to separate multiple usernames.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

AddHeader - Will add an arbitrary header line to the message, the entire line you wish to appear in the header must be specified. Multiple calls will add multiple headers.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

AddMessage - Will add the text to the currently stored body of the message. If there are attached files that have been already attached, calls to the AddMessage will add text after the file.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

AddTo - Will add the name string to the list of recipients on the 'To:' line. Commas may be used to separate multiple usernames.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

AttachFile - Will add the file named to the internal file storage. The file will be read in at the point of this call. The file may be uuencoded for transmittal.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

CsEmMailRelA - This is the default constructor. It will not fill in any of the user-settable fields.
Arguments:
Return Type:Void
Privilege:Public

Send - Will send the message in the current state. It will not delete any of the internally stored data.
Arguments:
Return Type:EcUtStatus
Privilege:Public

Subject - Will set the subject to the passed text string. Will delete any previously created subjects.
Arguments:str:EcTChar*
Return Type:EcUtStatus
Privilege:Public

~CsEmMailRelA - Will delete the object and all internal storage used. If mailing large files, it is recommended that this be performed as soon as possible.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The CsEmMailRelA class has associations with the following classes:
None

4.4.1.4 Dynamic Model Scenarios

4.4.1.4.1 Scenario #1

4.4.1.4.1.1 Abstract

In this scenario, a message is set in email.

4.4.1.4.1.2 Interfaces

The CsEmMailRelA class is used.

4.4.1.4.1.3 Stimulus

This class may be used when an email message must be sent.

4.4.1.4.1.4 Desired Response

After being called, a message should have been sent.

4.4.1.4.1.5 Participating Classes

CsEmMailRelA class is the only class involved.

4.4.1.4.1.6 Beginning State

No message has been sent.

4.4.1.4.1.7 Ending State

A message has been sent.

4.4.1.4.1.8 Scenario Description

In this scenario, an email message is sent to a user.

4.4.1.4.1.9 Event Trace

1. The object is instantiated.
2. AddTo() is called to name a recipient.
3. Subject() is called to specify a subject.
4. AddMessage() is called to specify a message body.
5. Send() is called to send the message.
6. The destructor for the object may then be called.

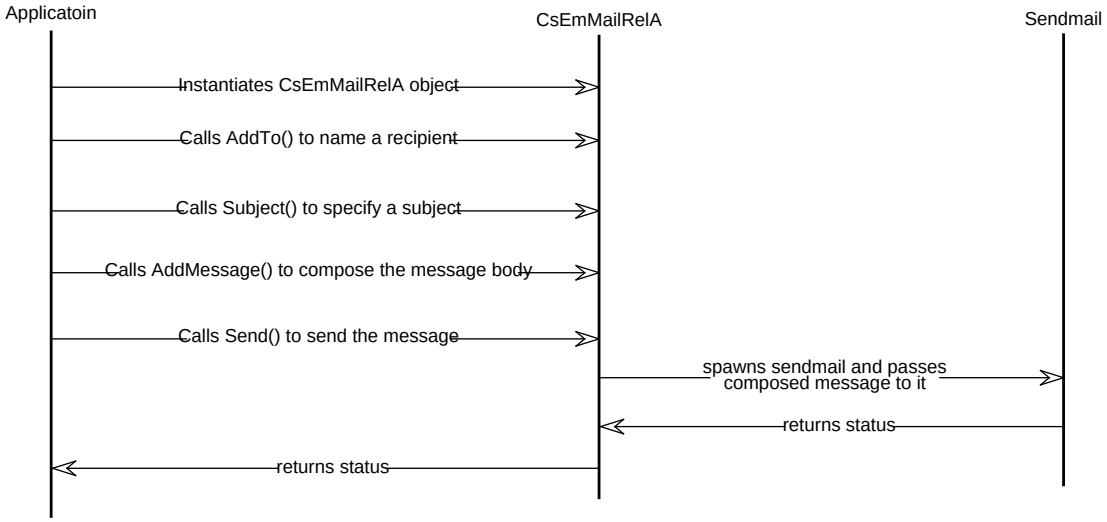

Figure 4.4.1.4-1. Email Service Event Trace

4.4.1.5 Functional Model and Implementation

Mail is created by having the object collect the information necessary to send the message. Upon receiving a send() it will spawn a sendmail process and use that to handle the actual delivery of the message.Aside from methods to set the standard header information there is an additional method worth mention. AddHeader will add an arbitrary text string to the header field - this allows future extensions. (For example some browsers make use of header information to indicate a MIME encoded message) Two threads should not manipulate the same mail object at the same time.

4.4.1.6 Service/CSCI Management and Strategy

4.4.1.6.1 System Management and Strategy

The sendmail program (distributed by the OS vendor) will need to be maintained as will the smtpd daemon, any email gateways necessary for the transit of the message, and the destination mail account if on our machines. For all kerberized connections, the Kerberos database and local files need to be maintained.

4.4.1.6.2 Operator Interface

There is no operator interface for the API, and the mail interfaces are defined by the COTS programs for the mail readers or by the OS for the daemons.

4.4.1.6.3 Reports

None.

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 CsFtFTPRelB object and send the object messages controlling a ftp session. CSS will also provide kerberized FTP for systems which will support it. In addition, CSS will provide the ability to execute a program for notification of successful pulls from the pull area. The CSS object is used to encapsulate FTP, an OTS standard application provided with most TCP/IP implementations.

4.4.2.3 Object Model

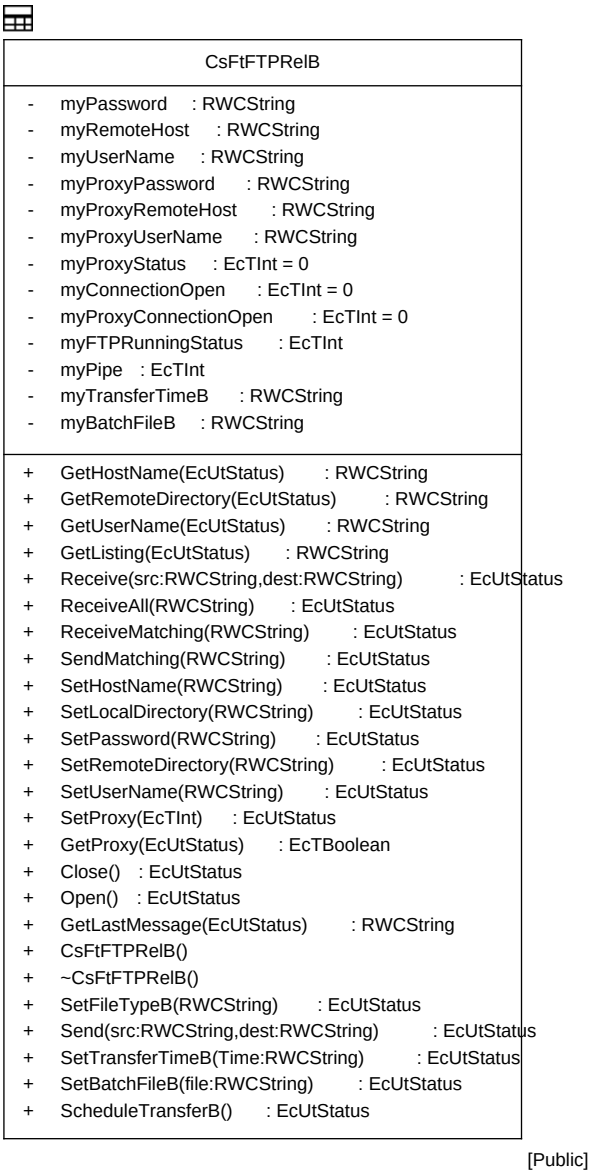

Figure 4.4.2.3-1. FTP Service Object Model

4.4.2.3.1 CsFtFTPRelB Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
CsFtFTPRelB provides an API for application programmer to initiates a FTP session to transfer files non-interactively. It also provides the capability to schedule file transfer.

Attributes:

myBatchFileB - used to store the batch script for batch mode transfer
Data Type:RWCString
Privilege:Private
Default Value:

myConnectionOpen - status bit to indicate if a connection is currently open on the primary host
Data Type:EcTInt
Privilege:Private
Default Value:0

myFTPRunningStatus - used to indicate the status of the pipe to the FTP program
Data Type:EcTInt
Privilege:Private
Default Value:

myPassword - used to store the password for the primary host to contact
Data Type:RWCString
Privilege:Private
Default Value:

myPipe - pipe for sending and receiving data from the FTP client program
Data Type:EcTInt
Privilege:Private
Default Value:

myProxyConnectionOpen - status bit to indicate if a connection is currently open on the proxy host
Data Type:EcTInt
Privilege:Private
Default Value:0

myProxyPassword - used to hold the password for the proxy connection
Data Type:RWCString
Privilege:Private
Default Value:

myProxyRemoteHost - used to hold the hostname for the proxy connection
Data Type:RWCString
Privilege:Private
Default Value:

myProxyStatus - used to hold a toggle bit to indicate if commands are directed to the primary host or the proxy host
Data Type:EcTInt
Privilege:Private
Default Value:0

myProxyUserName - used to hold the username for the proxy connection
Data Type:RWCString
Privilege:Private
Default Value:

myRemoteHost - used to store the hostname for the primary machine to contact
Data Type:RWCString
Privilege:Private
Default Value:

myTransferTimeB - used to store the time for batch mode transfer
Data Type:RWCString
Privilege:Private
Default Value:

myUserName - used to hold the username for the primary account
Data Type:RWCString
Privilege:Private
Default Value:

Operations:

Close - will close the connection
Arguments:
Return Type:EcUtStatus
Privilege:Public

CsFtFTPRelB - constructor for session
Arguments:
Return Type:Void
Privilege:Public

GetHostName - will return the host name of the connected client
Arguments:EcUtStatus
Return Type:RWCString
Privilege:Public

GetLastMessage - will return a pointer to a text buffer containing the exact text of the

message that was returned by the last command. The message may span multiple lines. The buffer will be over written when the next ftp command is called with the object, so copy the data to non-volital memory if you do not plan to inspect if before executing other command.
Arguments:EcUtStatus
Return Type:RWCString
Privilege:Public

GetListing - will return the file listing of the current directory on the remote mahcine
Arguments:EcUtStatus
Return Type:RWCString
Privilege:Public

GetProxy - will check the status of the proxy bit
Arguments:EcUtStatus
Return Type:EcTBoolean
Privilege:Public

GetRemoteDirectory - will return the working directory of the connected client
Arguments:EcUtStatus
Return Type:RWCString
Privilege:Public

GetUserName - will return the name of the current user
Arguments:EcUtStatus
Return Type:RWCString
Privilege:Public

Open - will open a new connection to a host and log the user on
Arguments:
Return Type:EcUtStatus
Privilege:Public

Receive - Will transfer the named file and store it in the file named in the optional second argument. A directory may not be specified as part of a path name (calls "get" command).
Arguments:src:RWCString,dest:RWCString
Return Type:EcUtStatus
Privilege:Public

ReceiveAll - will transfer all files on the remote machine in the current directory
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

ReceiveMatching - Will transfer all files on the remote machine in the current directory that match the specified as part of the wildcard.
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

ScheduleTransferB - will schedule the transfer
Arguments:
Return Type:EcUtStatus
Privilege:Public

Send - Will send the named file to the remote machine and store it in the file named specified in the optional second argument. (calls "put")
Arguments:src:RWCString,dest:RWCString
Return Type:EcUtStatus
Privilege:Public

SendMatching - Will transfer all files to the remote machine in the current directory that match the specified wildcard. A path may not be specified as part of the wildcard.
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

SetBatchFileB - Will set the batch script for scheduled transfer
Arguments:file:RWCString
Return Type:EcUtStatus
Privilege:Public

SetFileTypeB - Will set the file type for transfer (binary or ASCII)
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

SetHostName - Will set the name of the host to connect to.
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

SetLocalDirectory - Will execute the lcd command to change the local working directory
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

SetPassword - Will set the password to use to connect to the remote host
Arguments:RWCString

Return Type:EcUtStatus
Privilege:Public

SetProxy - Will set or unset the "proxy" bit. This will determine if future commands are set to the primary host or the proxy host. This prevents having two seperate identical sets of functions to send commands to the two servers. If this function is not called, the default value is 0 - which means that commands will go to the primary not the proxy host.
Arguments:EcTInt
Return Type:EcUtStatus
Privilege:Public

SetRemoteDirectory - Will execute the cd command to change the working directory on the remote machine
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

SetTransferTimeB - Will set the time for batch mode transfer
Arguments:Time:RWCString
Return Type:EcUtStatus
Privilege:Public

SetUserName - Will set the user name to be used with the FTP session
Arguments:RWCString
Return Type:EcUtStatus
Privilege:Public

~CsFtFTPRelB - destructor for session, will close all open connection
Arguments:
Return Type:Void
Privilege:Public

Associations:
The CsFtFTPRelB class has associations with the following classes:
None

4.4.2.4 Dynamic Model Scenarios

4.4.2.4.1 Scenario #1

4.4.2.4.1.1 Abstract

In this scenario, a program will instantiate a CsFtFTPRelB object and use it to transfer a file from a remote machine to the local machine. First it will instantiate a CsFtFTPRelB object and fill out the relevant information to open a connection (like hostname, username, etc..) After sending a

message to open the connection, the application will change directories, list files, and finally request the transfer of a file and close the connection.

4.4.2.4.1.2 Interfaces

CSS's CsFtFTPRelB object.

4.4.2.4.1.3 Stimulus

The CsFtFTPRelB object is instantiated when it is necessary to transfer a file between two machines.

4.4.2.4.1.4 Desired Response

The object, when used, should transfer the requested files and report any errors as they occur.

4.4.2.4.1.5 Participating Classes

CsFtFTPRelB is the only object involved in the transfers.

4.4.2.4.1.6 Beginning State

There is no connection open, and no files locally stored.

4.4.2.4.1.7 Ending State

There is no connection open, but there are now files locally stored.

4.4.2.4.1.8 Scenario Description

In this scenario, a client will retrieve a file found in a directory listing from a remote host, taking care to avoid overwriting any local files.

1. The application instantiates a CsFtFTPRelB object.
2. The application calls SetHostName() to set the host to connect to.
3. The application calls SetPassword() and SetUserName() to set password and username of caller.
4. The application calls open() which initiates the connection with the server.
5. The application calls SetRemoteDirectory() which passes the message through to the server and returns the status.
6. The application calls SetLocalDirectory() to change the local ftp directory.
7. The application calls GetListing() which returns a text message of all files which it uses to make sure that local files will not be over-written.
8. The application calls Receive() to transfer the file.
9. The application calls Close() to end the file transfer.
10. The application calls the object destructor which frees all memory associated with the object.

4.4.2.4.1.9 Event Trace

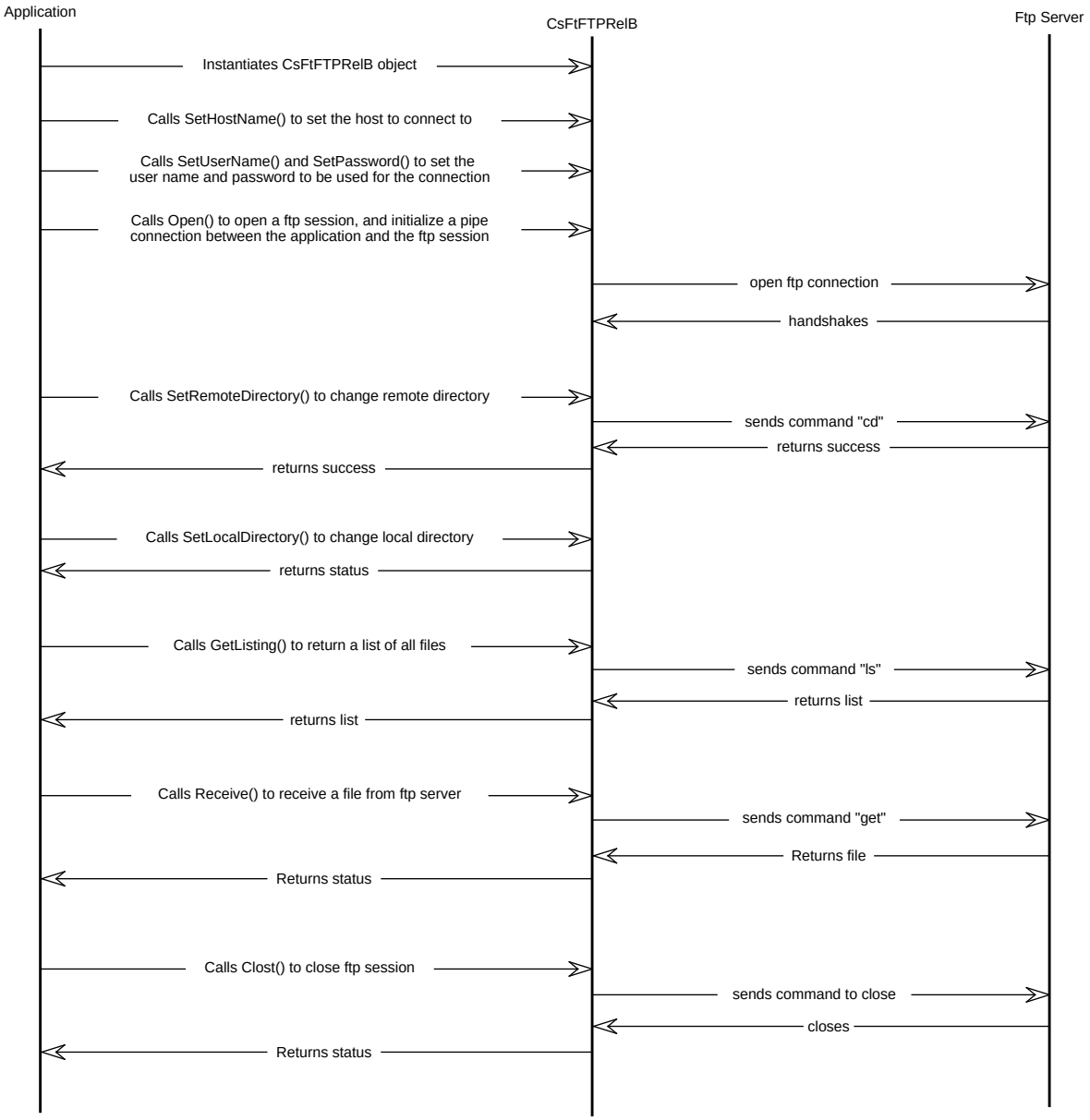

Figure 4.4.2.4-1. FTP Service Event Trace

4.4.2.4.2 Scenario #2

4.4.2.4.2.1 Abstract

In this scenario, a program will instantiate a CsFtFTPRelB object and use it to schedule a file transfer. First it will instantiate a CsFtFTPRelB object and fill out the relevant information for batch mode transfer (transfer time and batch file, which will hold each of the commands to the ftp connection) The application will then execute the batch file to perform file transfer at the specify time, and finally close the connection.

4.4.2.4.2.2 Interfaces

CSS's CsFtFTPRelB object.

4.4.2.4.2.3 Stimulus

The CsFtFTPRelB object is instantiated when it is necessary to schedule a file transfer between two machines.

4.4.2.4.2.4 Desired Response

The object, when used, should log results of batch mode transfer to a specified log files. In case of scheduled operation fails, it should generate events using event logging service.

4.4.2.4.2.5 Participating Classes

CsFtFTPRelB is the only object involved in the transfers.

4.4.2.4.2.6 Beginning State

There is no connection open, and no files locally stored.

4.4.2.4.2.7 Ending State

There is no connection open, but there are now files locally stored.

4.4.2.4.2.8 Scenario Description

In this scenario, a client will schedule a file transfer for a specific time.

1. The application instantiates a CsFtFTPRelB object.
2. The application calls SetTransferTime() to set the time to perform transfer.
3. The application calls SetBatchFile() to set the batch file which will hold each of the commands to the ftp connection. SetBatchFile() accept an optional argument filename. If filename is provided, this file will be used to hold each of the ftp commands. This method will also set the batch mode transfer flag to be true. If filename is not provided, a default file will be used.
4. The application will call each of the commands that need to be performed for the scheduled transfer. The application calls SetHostName() to set the host to connect to.
5. The application calls SetPassword() and SetUserName() to set password and username of caller.

6. The application calls Open() which initiates the connection with the server.
7. The application calls SetRemoteDirectory() which passes the message through to the server and returns the status.
8. The application calls Receive() to transfer the file.
9. The application calls Close() to end the file transfer.
10. The application calls ScheduleTransfer() to put the scheduled transfer in queue, and execute the transfer at the specified time.
11. The application calls the object destructor which frees all memory associated with the object.

4.4.2.4.2.9 Event Trace

See Figure

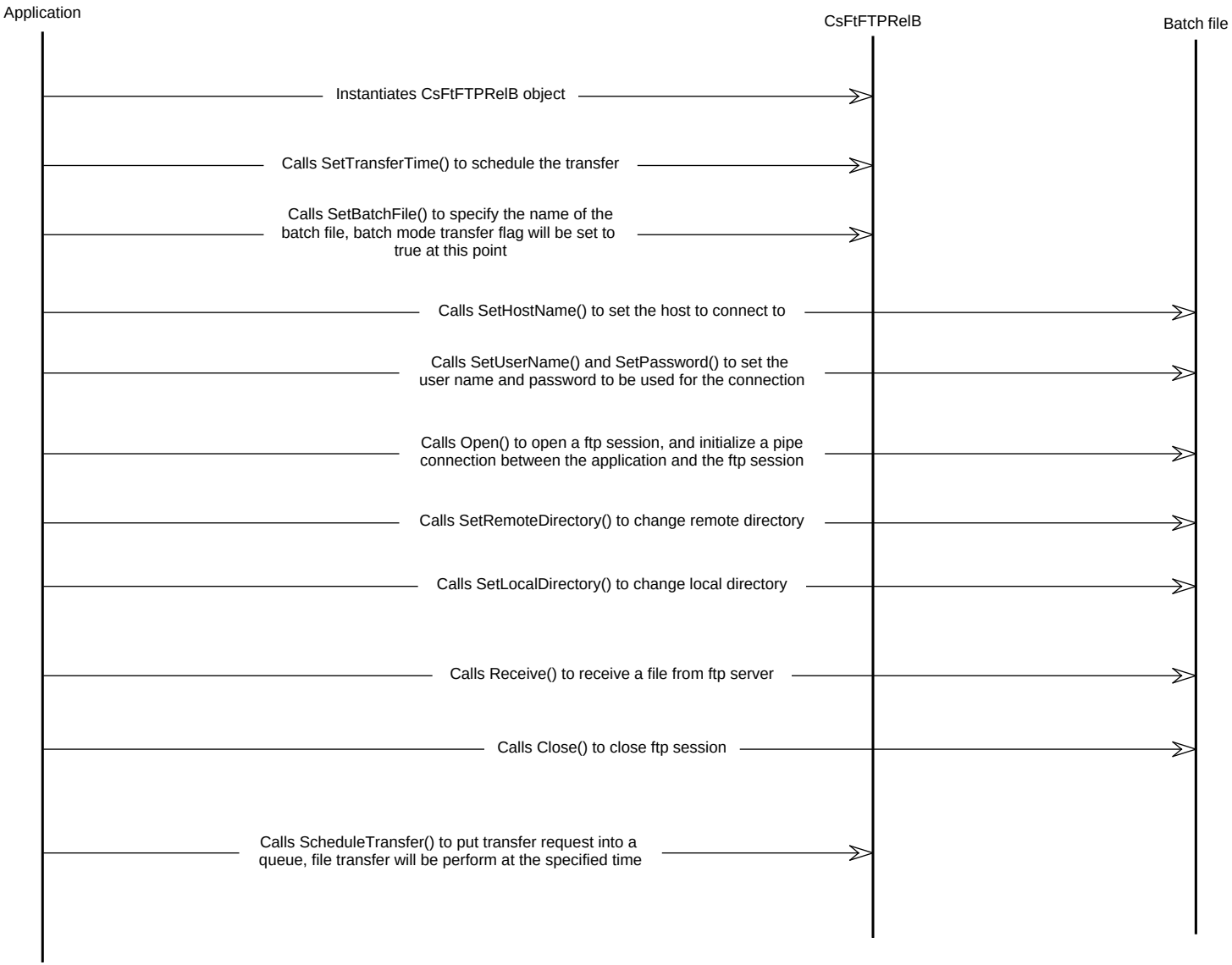

Figure 4.4.2.4-2. FTP Service Event Trace

4.4.2.5 Functional Model and Implementation

There are two basic separate parts in the implementation. For the client API:

The object is implemented fairly simply. Upon receiving the open command, the object will spawn an independent ftp client process. It will then use a created pipe to send requests to the ftp client, and will return any errors or data to the caller of the method. Thus, if a given command, say a 'change directory', fails, the application will receive an error from that function. It will also have a non-interactive mode for which it will create a script which it will schedule to run at the appropriate time. Upon deletion of the object, or a call to the close() method, the object will close the connection, and in the case of a non-interactive job, it will schedule the transfer for later. FTP servers and clients are standard on unix boxes and do not require any changes. Multiple threads may not simultaneously send commands to the same CsFtFTPRelB object. If an unexpected error occurs (like the ftp client dying, or the connection being unexpectedly lost) then an exception will be thrown to the application. To verify that a transfer completed successfully, the application will need to call the GetLastMessage() function call to check the text string that the ftp server used to reply to the client to make sure it is in agreement with what was expected.

For the FTP server modifications are necessary to the OTS code to notify the dataserver of a successful pull:

The ftpd server will be modified to spawn a new process and execute a provided program each time a file is transferred. The arguments to the executed program would be as follows:

joeuser 784953937 pull myfile.dat

which would indicate that user 'joeuser' successfully pulled file 'myfile.dat' at the time that is 784953937 seconds since Jan. 1, 1970. (which happens to be Nov. 15 21:45:37 1994) This representation of time is a well known standard for unix.

4.4.2.6 Service/CSCI Management and Strategy

4.4.2.6.1 System Management and Strategy

Not Applicable.

4.4.2.6.2 Operator Interface

There is no operator interface since the class only provides an interface for existing services. However, the operators will have to maintain the local ftp client, the remote ftpd server, and the remote inetd which spawns ftpd when needed. In addition, for the pull area, the customized executable for notification will need to be maintained.

4.4.2.6.3 Reports

None.

4.4.3 DFS

4.4.3.1 Overview

Distributed File Service (DFS) provides location independent file access with high performance, availability and security. With the remote file access (RFA) capabilities provided by DFS, users

can access remote files as if they are on the local file system. The whole process is transparent - upon authentication at login with the file servers up, no difference is noticed between a DFS file and files on local disk.

4.4.3.2 Context

DFS is a distributed DCE application. As such, ECS can derive benefits from its integration with and use of DCE services. Within the ECS system users and programs frequently need to share files. This is especially true in the case of individual DAACs. Against such a backdrop DFS makes information sharing easy because everyone in a cell has access to one seamless file system. In addition, DFS builds on DCE functionality to provide a single global integrated file system to the entire ECS system.

4.4.3.3 Object Model

No object model is provided for DFS.

4.4.3.4 Dynamic Model Scenarios

4.4.3.4.1 Scenario #1

4.4.3.4.1.1 Abstract

This scenario describes the interaction of a science user with DFS. The user, using a visualization tool, intends to examine the weather data for a specific date which is bound as one set per month. This requires the user to repeatedly access the same data over a certain length of time. The data itself is contained in a specific part of a large data product which is indexed and resides in common area mounted on DFS.

4.4.3.4.1.2 Interfaces

The visualization tool being used by the scientist.

4.4.3.4.1.3 Stimulus

File access required by the visualization tool.

4.4.3.4.1.4 Desired Response

Access to the file is provided/denied based on authorization checking. Upon successful authorization the file is made available to the user from the file server. In addition, the specific portion of the data which the user wishes to view is cached by DFS in the local system.

4.4.3.4.1.5 Participating Classes

Not Applicable.

4.4.3.4.1.6 Beginning State

The requested file is not locally stored. User is an authenticated DCE principal. In addition, the Data Server has already deposited the data product as a file in a common area mounted on DFS.

4.4.3.4.1.7 Ending State

Part of the file is cached to local machine.

4.4.3.4.1.8 Scenario Description

1. Using the interface provided by the visualization tool, the user changes to the desired directory and chooses the file he wishes to access. Note that to get to a DFS directory in another cell, the complete path name must be provided.
2. The DFS server examines the users identity and group membership and determines if the access requested is allowed. If okay, success is returned.
3. The user indicates the specified part of data using the interface provided by the visualization tool.
4. DFS caches this portion of the data on the local machine using the cache manager.
5. The user repeatedly displays the data using the visualization tool. This repeated access to the same part of the file does not result in any DFS related traffic.

4.4.3.4.1.9 Event Trace

4.4.3.4.2 Scenario #2

4.4.3.4.2.1 Abstract

A user wishes to read and modify a file stored at some DAAC from a personal workstation. This access must appear to be a standard local access. When done, the user will save his/her changes.

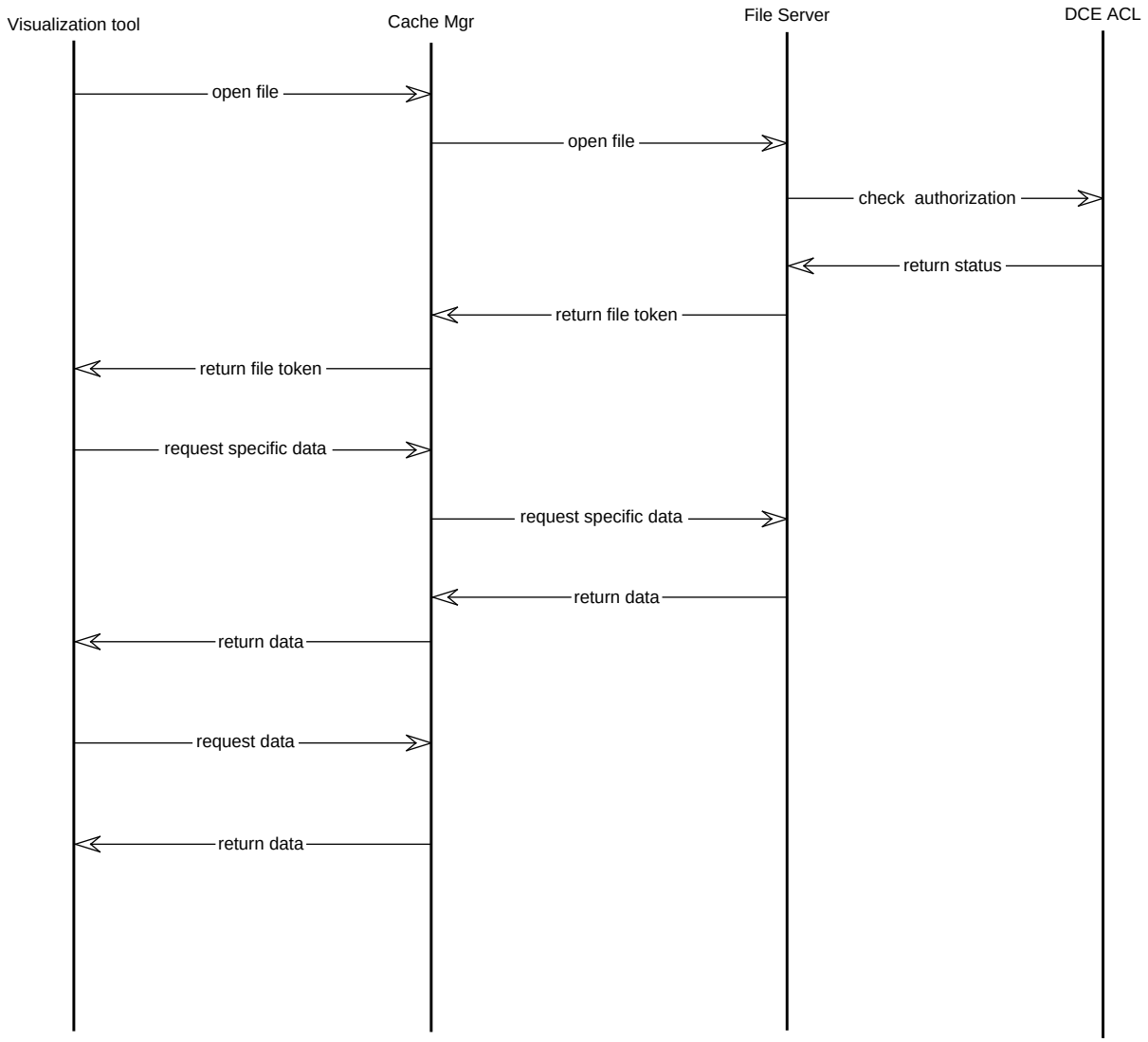

Figure 4.4.3.4-1. Event Trace for DFS Science User Access (Release B)

4.4.3.4.2.2 Interfaces

The editor.

4.4.3.4.2.3 Stimulus

The user opens the file in a favorite editor by specifying a Unix path name to the file.

4.4.3.4.2.4 Desired Response

The file will be loaded into the editor and the user can read and modify the file like a normal file. Changes will be visible to other users after the file is closed.

4.4.3.4.2.5 Participating Classes

None.

4.4.3.4.2.6 Beginning State

User is an authenticated DCE principal.

4.4.3.4.2.7 Ending State

Changes are stored in DFS for other users.

4.4.3.4.2.8 Scenario Description

1. User selects open on a file stored in DFS from within her favorite editor.
2. The editor uses the standard Unix "open" call specifying the path name to the file.
3. After detecting that the file is in DFS, the OS invokes the cache manager API for open.
4. The Cache Manager contacts the DFS FLDB to locate the fileset which contains the file in question. This may take several steps since DFS resolves the path one piece at a time.
5. The DFS server examines the users identity and group membership and determines if the access requested is allowed. If okay, success is returned.
6. As the user accesses the file, the editor will issue the Unix read command. The Cache Manager will fetch pieces of the file on demand from the DFS server and return them to the editor.
7. The user makes some changes to the file and saves it.
8. The editor will issue a write call.
9. The Cache Manager will obtain a write token from DFS and then save the changes locally.
10. The user closes the file causing the editor to call "close".
11. The Cache Manager sends file changes to the DFS server which stores them.

4.4.3.4.2.9 Event Trace

4.4.3.4.3 Scenario #3

4.4.3.4.3.1 Abstract

A file server fails while files on it are being read.

4.4.3.4.3.2 Interface

A program that reads the file.

4.4.3.4.3.3 Stimulus

File server fails while file is being read.

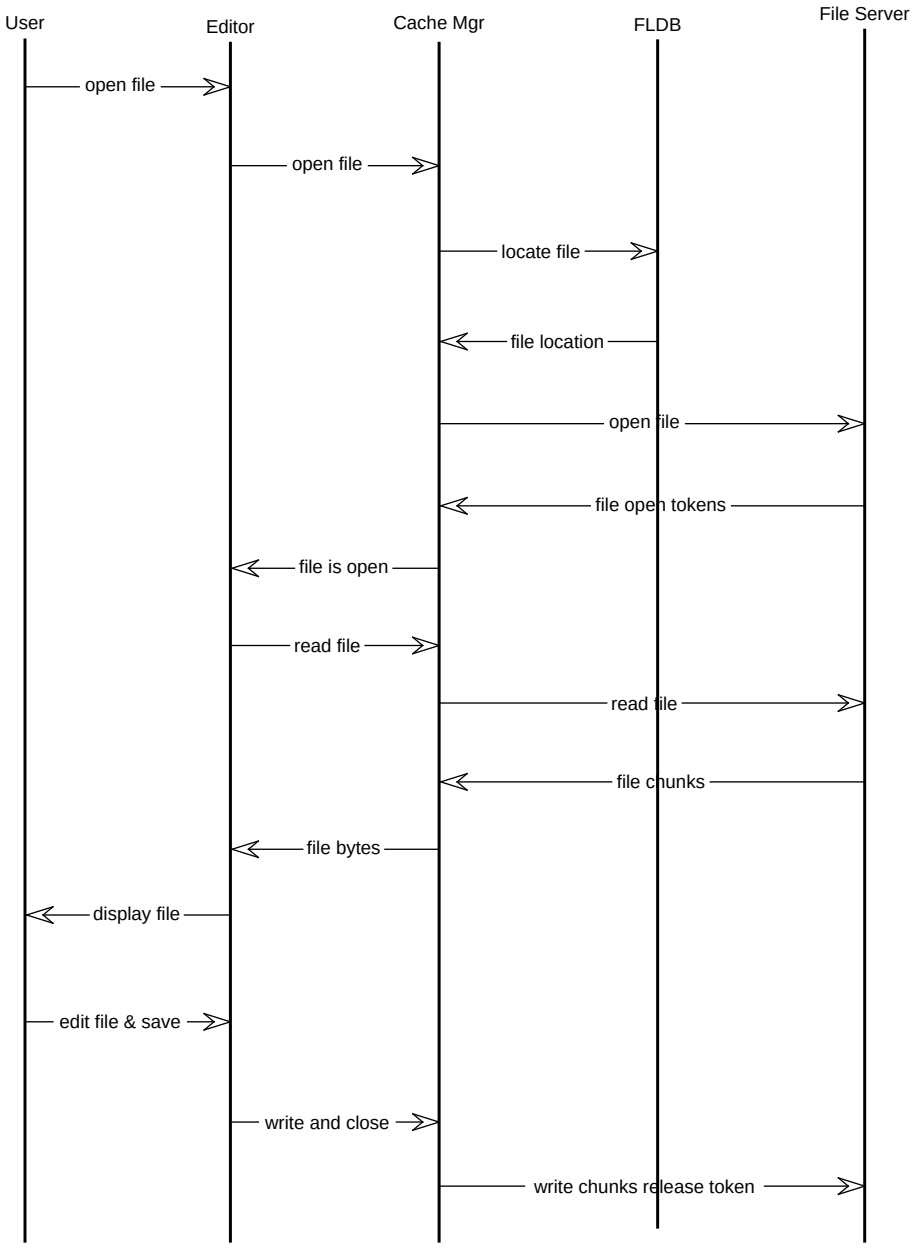

Figure 4.4.3.4-2. Event Trace for DFS (Release B)

4.4.3.4.3.4 Desired Response

Readers of the file will continue to have access. This will be transparent.

4.4.3.4.3.5 Participating Classes

None.

4.4.3.4.3.6 Beginning State

Files in fileset a are open for reading on server A. The filesets on A are replicated to server B. In particular, a is replicated as fileset a'. The FLDB must also be replicated.

4.4.3.4.3.7 Ending State

Clients should be unaffected.

4.4.3.4.3.8 Scenario Description

1. Client access file from fileset a on server A.
2. Client cache manager fetches chunks of file from A.
3. Server A fails.
4. Client accesses more parts of files from fileset a.
5. Cache Manager contacts server A for chunks.
6. After a time-out, the Cache Manager determines that A is unavailable.
7. The Cache Manager contacts the FLDB to obtain a new server that holds fileset a.
8. The FLDB returns the file server B.
9. The Cache Manager contacts server B and requests chunks from fileset a'.
10. The chunks are returned, and operation continues as normal.

4.4.3.4.3.9 Event Trace

4.4.3.4.4 Scenario #4

4.4.3.4.4.1 Abstract

A system administrator must move a fileset from one server to another.

4.4.3.4.4.2 Interfaces

DFS administrative tool.

4.4.3.4.4.3 Stimulus

The system administrator issues the DFS move fileset command.

4.4.3.4.4.4 Desired Response

The fileset will move to a new server AND users will be unaware of the transfer.

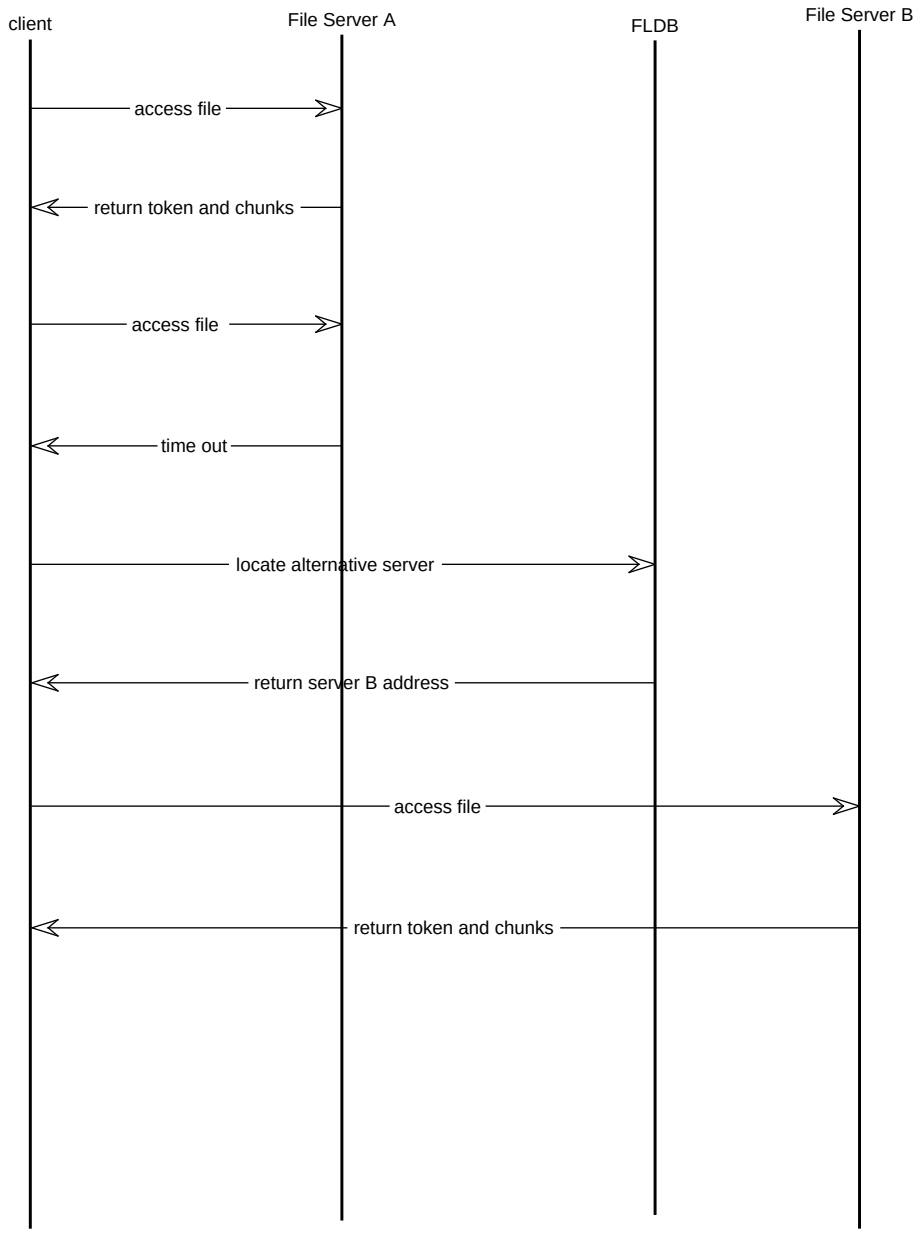

Figure 4.4.3.4-3. Event Trace for DFS Server Failure Scenario (Release B)

4.4.3.4.4.5 Participating Classes

None.

4.4.3.4.4.6 Beginning State

Fileset 1 is on server A.

4.4.3.4.4.7 Ending State

Fileset 1 is on server B. Users are unaware of move.

4.4.3.4.4.8 Scenario Description

1. Users are accessing files in fileset 1.
2. Sysadmin issues fileset move command.
3. DFS begins cloning fileset 1 by copying its contents to server B WITHOUT locking anything.
4. After the first pass, DFS copies the changes made during the first pass.
5. After the second pass, DFS locks the fileset (preventing write access) and copies the changes made during the second pass. This operation should be very brief. Users may experience a short delay in file writes.
6. DFS updates the FLDB entry for fileset 1 to point to server B.
7. Some user attempts to access a file from fileset 1.
8. The Cache Manager will be unable to access the fileset from server A. It will instead contact the FLDB to obtain the fileset's new location.
9. The Cache Manager will access the file from server B.

4.4.3.4.4.9 Event Trace

4.4.3.5 Functional Model and Implementation

Not Applicable

4.4.3.6 Service/CSCI Management and Strategy

4.4.3.6.1 System Management and Strategy

The amount of administration needed to maintain DFS depends largely on the number of files stored in DFS. Some of the factors that can affect system management and administration include, initial configuration choices, physical location of the file servers, load balancing between fileservers, familiarity of the operators with local user needs and policies related to replication and backups/restores. System management includes monitoring and adjusting the file system performance, using access control lists to control access to files and directories and using administrative lists to control DFS server access (to prevent unauthorized principals from tinkering with DFS servers). In general, DFS attempts to manage itself where possible. The DFS BOS (Basic OverSeer) monitors and restarts DFS servers as needed. Finally, DFS includes integrated backup facilities to aid in the management of fileset backups.

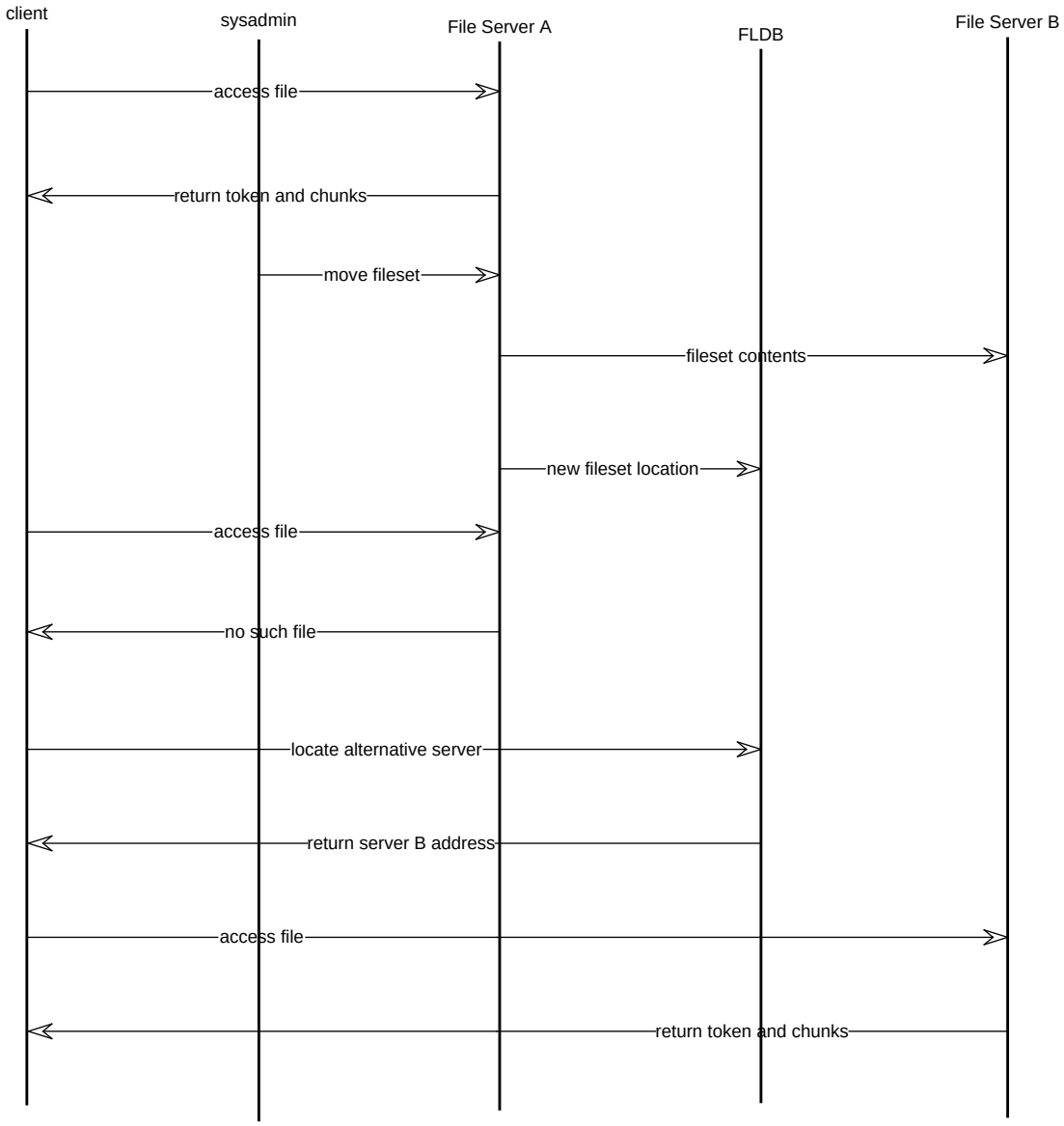

Figure 4.4.3.4-4. Event Trace for DFS-System Administrator Interaction (Release B)

4.4.3.6.2 Operator Interface

DFS provides a standard command line interface which can be used for several administrative functions. Furthermore, DFS being a a major subsystem within DCE some portions of its management are performed by using the standard DCE management interfaces to CDS and DCE Security. In addition, DFS comes with several command line tools that allow administrators to manage filesets, access control, and the various DFS servers.

4.4.3.6.3 Reports

Not Applicable.

4.4.3.7 Frequently Asked Questions

1) What is a distributed file system?

   A distributed filesystem is a filesystem in which access to remote files is the same as to local files.

2) What are the benefits of having a distributed file system?

   Without a distributed filesystem, if a user on one host wanted to access file on another host, he would need to use some special utility. For example, on Unix systems, rlogin, telnet or ftp could possibly be used. On the other hand with a distributed filesystem, no special procedures or utilities need to be used and the user would not typically know that the file being accessed is remote. One of the most popular distributed filesystems today is the Network File System (NFS). The DCE Distributed File Service (DFS) is a newer distributed filesystem from OSF and is rapidly becoming very popular.

3) What are the advantages of DFS?

   Distributed File Service (DFS) provides location independent file access with high performance, availability and security. With the remote file access (RFA) capabilities provided by DFS, users can access remote files as if they are on the local file system.

   Some advantages of DFS include:

   - Uniform file access through uniform name space
   - Full DCE based authentication and ACL authorization
   - Local caching for high performance
   - Replication for high availability and performance
   - Logging file system for rapid server recovery
   - Tools for management and administration control
   - Gateway for Network File System (NFS) clients

4) What is a fileset?

   A fileset is a hierarchical grouping of files and directories managed as a single unit. A fileset can be any size but is always smaller than a single UNIX disk partition. Unlike UNIX partitions, filesets can be easily expanded or moved. They can also be replicated to improve availability.

5) How does replication work?

   Replication creates multiple read only copies of a DFS fileset and places them on multiple server machines. As a result, the failure of a single machine that is housing a replicated fileset will not interrupt read only access to that fileset. This increases file system availability.

6) How does DFS support location independence?

   DFS tracks the location of every individual fileset. This information is maintained in the Fileset Location Data Base (FLDB). When DFS clients need access to a fileset, they request its current location from the FLDB. As a result, filesets can be moved any time without impacting access to the fileset. The FLDB is replicated to ensure that it is always available. By tracking fileset location in the FLDB, administrators are freed from having to record the location of every fileset.

7) How does DFS interoperate with NFS?

   The DFS/NFS Secured Gateway allows NFS clients to securely access the DFS filespace. A Gateway Server runs on a machine which is DFS client and exports the DFS filespace (/...) via NFS. The Gateway Server appears to be a standard NFS file server to NFS clients.

8) What does the client system need to access DFS?

   A DFS client machine requires a DCE secure core, the DFS client daemon (dfsd) as well as a modified kernel that incorporates DFS calls. In addition, space must be allocated on the local file system for the DFS cache. The cache can be of any size, although a larger cache will allow the DFS Cache Manager to cache more information and thus improve performance.

9) How does DFS ensure secure file access?

   DFS is fully integrated with DCE. As such it takes full advantage of the DCE security service. DFS users must be authenticated DCE principals. Each access is checked using DCE ACLs. These ACLs can be set by users on a file or directory basis. The allowed permissions are read, write, execute, control, insert and delete. ACLs allow individual users to receive from the file server permission to perform particular operations on files or directories that are stored in a file server.

4.4.4 Bulletin Board

4.4.4.1 Overview

This service provides interactive (for operators) and development (API, for applications) interfaces to manage bulletin board messages. Interactive interfaces provide a full set of functionality, and are entirely COTS. The API is limited in scope as it only addresses sending a message from within an application.

4.4.4.2 Context

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

4.4.4.3 Object Model

The object Model for Bulletin Board service is combined with Email service object Model. Refer to Section 4.4.1.3.

4.4.4.4 Dynamic Model Scenarios

4.4.4.4.1 Scenario #1

4.4.4.4.1.1 Abstract

In this scenario, a message is posted to a public bulletin board.

4.4.4.4.1.2 Interfaces

The CsBBMailRelA class is used for the interface.

4.4.4.4.1.3 Stimulus

At times a message will need to be posted publicly. This class will allow such posting.

4.4.4.4.1.4 Desired Response

After the class is called a message should be posted.

4.4.4.4.1.5 Participating Classes

The CsBBMailRelA class and the CsEmMailRelA (used transparently) class are used. The entire interface is through the CsBBMailRelA class, however, the underlying structure has CsBBMailRelA inheriting many methods and storage variables from CsEmMailRelA to prevent redundant code being written.

4.4.4.4.1.6 Beginning State

In the beginning, no message has been posted.

4.4.4.4.1.7 Ending State

After using the object, a message will be posted to the appropriate bulletin board.

4.4.4.4.1.8 Scenario description

Sending a message to a bulletin board looks nearly identical to sending an email message. The only difference is that an nntp server needs to be specified.

4.4.4.4.1.9 Event Trace

1. The object is instantiated.
2. AddTo() is called to name a destination bulletin board (ex "comp.soft-sys.dce").
3. Subject() is called to specify a subject.
4. SetNNTPHost() is called to specify the host to post the message to.
5. AddMessage() is called to add text to the message body.
6. Send() is called to send the message.
7. The destructor for the object may then be called.

4.4.4.5 Functional Model and Implementation

CSS is providing a class which contains APIs to allow posting of messages. This object inherits most of the data structures and methods from the CsBBMailRelA object used for sending email. A nntp host has been added and a few non-applicable methods have been overridden. Two threads should not manipulate the same mail object at the same time - the results are not defined.

4.4.4.6 Service/CSCI Management and strategy

4.4.4.6.1 System Management and Strategy

The nntp server will need to be maintained outside of CSS.

4.4.4.6.2 Operator Interface

There is no operator interface for the API, and the bulletin board interfaces are defined by the COTS programs for the readers and servers.

4.4.4.6.3 Reports

None.

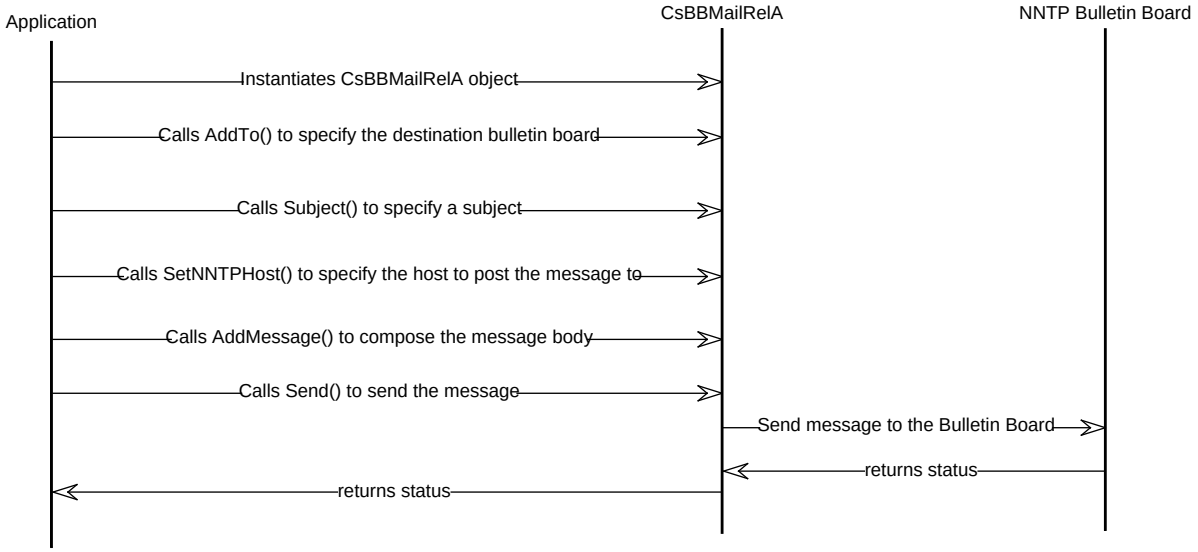

Figure 4.4.4.4-1. RelB_BB_EventTrace Dynamic Model

4.4.5 Virtual Terminal

4.4.5.1 Overview

Virtual terminal (VT) hides the terminal characteristics and handling conventions from both the operator and server host by allowing both parties to deal with a virtual device that has similar capabilities. VT provides operators the capability to remotely log onto ECS machines. CSS will also provide kerberized telnet and telnetd on available systems.

In addition, CSS will provide common facility support for the ECS dial-up service (Please refer to 305-CD-004--001).

4.4.5.2 Context

CSS is providing kerberized telnet and telnetd for systems where available. Telnet and telnetd (non-kerberized) are distributed as part of the OS.

The dial-up service provides user access to the ECS search and order tool character-based user interface (CHUI).

4.4.5.3 Object Model

There are no objects involved with a Virtual Terminal.

4.4.5.4 Dynamic Model Scenarios

4.4.5.4.1 Scenario #1

Not Applicable.

4.4.5.4.1.1 Abstract

Not Applicable.

4.4.5.4.1.2 Interfaces

Not Applicable.

4.4.5.4.1.3 Stimulus

Not Applicable.

4.4.5.4.1.4 Desired Response

Not Applicable.

4.4.5.4.1.5 Participating Classes

Not Applicable.

4.4.5.4.1.6 Beginning State

Not Applicable.

4.4.5.4.1.7 Ending State

Not Applicable.

4.4.5.4.1.8 Scenario Description

Not Applicable.

4.4.5.4.1.9 Event Trace

None

4.4.5.5 Functional Model and Implementation

The implementation will be telnet clients and servers - these are completely COTS and provided by OS or public domain.

The infrastructure needed for the dial-up service is being provided by ISS. Specific details in this regard may be obtained from the ISS design document. CSS implementation of the service will include providing the necessary support to ISS in designing specialized user accounts that may be needed for the dial-up users to access ECS.

4.4.5.6 Service/CSCI Management and Strategy

4.4.5.6.1 System Management and Strategy

The server binary needs to be maintained, and it is started dynamically by the process inetd which also needs to be maintained. For all kerberized connections, the Kerberos database and local files need to be maintained.

4.4.5.6.2 Operator Interface

While there is no direct operator interface provided by CSS since all software is COTS, the telnetd and inetd process will need to be maintained to allow new connections, and the telnet client will need to be maintained.

4.4.5.6.3 Reports

None.

4.4.6 Event Logging

4.4.6.1 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.

4.4.6.2 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.

4.4.6.3 Object Model

EcUtLoggerRelA object for application logging.

EcUtLoggerRelADebug object for logging application debug messages.

EcUtLoggerRelAMgmt object (abstract) for different management events. The following objects inherit from the EcUtLoggerRelAMgmt object:

EcUtLoggerRelAFault object for logging Fault events.

EcUtLoggerRelAPerf object for logging Performance events.

EcUtLoggerRelAAudit object for logging Audit events.

EcUtLoggerRelASec object for logging Security events.

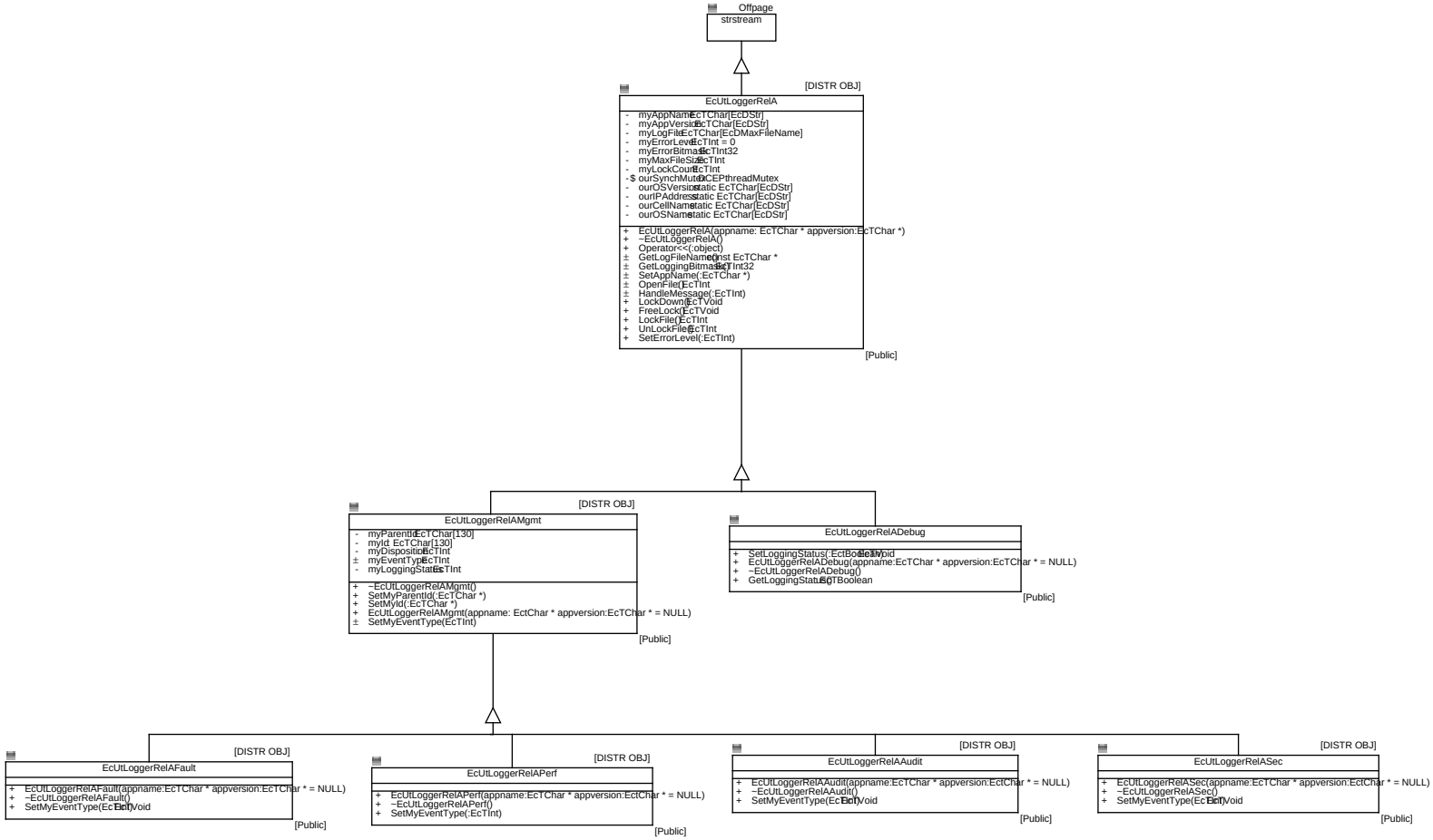

Figure 4.4.6.3-1. Event Logging Service Object Model

4.4.6.3.1 EcUtLoggerRelA Class

Parent Class:strstream
Public:Yes
Distributed Object:Yes
Purpose and Description:
This service allows applications to log event and history information to a file which can later be used for study.

Attributes:

myAppName - will be used to hold the application name
Data Type:EcTChar[EcDStr]
Privilege:Private
Default Value:

myAppVersion - will be used to hold the version number
Data Type:EcTChar[EcDStr]
Privilege:Private
Default Value:

myErrorBitmask - will be used to hold the ErrorLevel bitmask. It will determine which error levels will get logged.
Data Type:EcTInt32
Privilege:Private
Default Value:

myErrorLevel - will be used to hold the currently set error level this has a default level of 0
Data Type:EcTInt
Privilege:Private
Default Value:0

myLockCount - will be used in conjunction with the thread writing mutex to keep a count of locks to the recursive mutex.
Data Type:EcTInt
Privilege:Private
Default Value:

myLogFile - will be used to hold the name of the log file.
Data Type:EcTChar[EcDMaxFileName]
Privilege:Private
Default Value:

myMaxFileSize - Will hold the maximum file size of the file.
Data Type:EcTInt
Privilege:Private
Default Value:

ourCellName - will hold the cell name.
Data Type:static EcTChar[EcDStr]
Privilege:Private
Default Value:

ourIPAddress - will be used to hold the ip address of the local machine
Data Type:static EcTChar[EcDStr]
Privilege:Private
Default Value:

ourOSName - will hold the OS name
Data Type:static EcTChar[EcDStr]
Privilege:Private
Default Value:

ourOSVersion - will be used to hold the os name and version
Data Type:static EcTChar[EcDStr]
Privilege:Private
Default Value:

ourSynchMutex - mutex to protect internal integrity.
Data Type:DCEPthreadMutex
Privilege:Private
Default Value:
This is a Class Attribute.

Operations:

EcUtLoggerRelA - Constructor. Will use the appname to determine the filename to log to. The appversion is optional or may be NULL.
Arguments:appname: EcTChar * appversion:EcTChar *
Return Type:Void
Privilege:Public

FreeLock - - internal function - frees lock
Arguments:
Return Type:EcTVoid
Privilege:Public

GetLogFileName - Will return the log file name to the application. No guarantee is made about the format of the file.
Arguments:
Return Type:const EcTChar *
Privilege:Protected

GetLoggingBitmask - Will return the bitmask of which errors are currently being logged.
Arguments:
Return Type:EcTInt32
Privilege:Protected

HandleMessage - Used to handle control messages
Arguments::EcTInt
Return Type:Void
Privilege:Protected

LockDown - - internal function - locks down structures
Arguments:
Return Type:EcTVoid
Privilege:Public

LockFile - - internal function - locks file
Arguments:
Return Type:EcTInt
Privilege:Public

OpenFile - - internal function - opens file
Arguments:
Return Type:EcTInt
Privilege:Protected

Operator<< - will be used to send messages to the object. Chaining of <<s is encouraged.
Arguments::object
Return Type:Void
Privilege:Public

SetAppName - Will set the current application - and will as a result have its log file determined.
Arguments::EcTChar *
Return Type:Void
Privilege:Protected

SetErrorLevel - Will set the error level for the current message.
Arguments::EcTInt
Return Type:Void

Privilege:Public

UnLockFile - - internal function - locks file
Arguments:
Return Type:EcTInt
Privilege:Public

~EcUtLoggerRelA - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelA class has associations with the following classes:
None

4.4.6.3.2 EcUtLoggerRelAAudit Class

Parent Class:EcUtLoggerRelAMgmt
Public:Yes
Distributed Object:Yes
Purpose and Description:
Used to create Audit event logging objects.

Attributes:

All Attributes inherited from parent class

Operations:

EcUtLoggerRelAAudit - constructor - accepts application name and an optional application version as arguments.
Arguments:appname:EcTChar * appversion:EctChar * = NULL
Return Type:Void
Privilege:Public

SetMyEventType - Sets the event type.
Arguments:EcTInt
Return Type:EcTVoid
Privilege:Public

~EcUtLoggerRelAAudit - destructor
Arguments:

Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelAAudit class has associations with the following classes:
None

4.4.6.3.3 EcUtLoggerRelADebug Class

Parent Class:EcUtLoggerRelA
Public:Yes
Distributed Object:No
Purpose and Description:
Used to log application debugging information.

Attributes:

All Attributes inherited from parent class

Operations:

EcUtLoggerRelADebug - Constructor. Accepts as arguments the application name and an optional version string.
Arguments:appname:EcTChar * appversion:EcTChar * = NULL
Return Type:Void
Privilege:Public

GetLoggingStatus - Will return if logging is currently on or off.
Arguments:
Return Type:EcTBoolean
Privilege:Public

SetLoggingStatus - This will turn on and off logging of messages.
Arguments::EctBoolean
Return Type:EcTVoid
Privilege:Public

~EcUtLoggerRelADebug - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelADebug class has associations with the following classes:
None

4.4.6.3.4 EcUtLoggerRelAFault Class

Parent Class:EcUtLoggerRelAMgmt
Public:Yes
Distributed Object:Yes
Purpose and Description:
Used to log Fault events.

Attributes:

All Attributes inherited from parent class

Operations:

EcUtLoggerRelAFault - constructor - accepts application name and an optional application version as arguments.
Arguments:appname:EcTChar * appversion:EcTChar * = NULL
Return Type:Void
Privilege:Public

SetMyEventType - Used to set the event type.
Arguments:EcTInt
Return Type:EcTVoid
Privilege:Public

~EcUtLoggerRelAFault - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelAFault class has associations with the following classes:
None

4.4.6.3.5 EcUtLoggerRelAMgmt Class

Parent Class:EcUtLoggerRelA
Public:Yes
Distributed Object:Yes

Purpose and Description:
Virtual object for management logging.

Attributes:

myDisposition - will hold the disposition
Data Type:EcTInt
Privilege:Private
Default Value:

myEventType - will hold the event type
Data Type:EcTInt
Privilege:Protected
Default Value:

myId - will hold id
Data Type:EcTChar[130]
Privilege:Private
Default Value:

myLoggingStatus - Logging status
Data Type:EcTInt
Privilege:Private
Default Value:

myParentId - Will hold parent id
Data Type:EcTChar[130]
Privilege:Private
Default Value:

Operations:

EcUtLoggerRelAMgmt - constructor - accepts as arguments, the application name and an opional version.
Arguments:appname: EctChar * appversion:EcTChar * = NULL
Return Type:Void
Privilege:Public

SetMyEventType - Used to set the event type.
Arguments:EcTInt
Return Type:Void
Privilege:Protected

SetMyId - used to set the id
Arguments::EcTChar *
Return Type:Void
Privilege:Public

SetMyParentId - used to set the parent id
Arguments::EcTChar *
Return Type:Void
Privilege:Public

~EcUtLoggerRelAMgmt - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelAMgmt class has associations with the following classes:
None

4.4.6.3.6 EcUtLoggerRelAPerf Class

Parent Class:EcUtLoggerRelAMgmt
Public:Yes
Distributed Object:Yes
Purpose and Description:
Object for logging Performance events.

Attributes:

All Attributes inherited from parent class

Operations:

EcUtLoggerRelAPerf - Constructor.
Arguments:appname:EcTChar * appversion:EctChar * = NULL
Return Type:Void
Privilege:Public

SetMyEventType - Set the event type.
Arguments::EcTInt
Return Type:Void
Privilege:Public

~EcUtLoggerRelAPerf - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelAPerf class has associations with the following classes:
None

4.4.6.3.7 EcUtLoggerRelASec Class

Parent Class:EcUtLoggerRelAMgmt
Public:Yes
Distributed Object:Yes
Purpose and Description:
Used to log Security events.

Attributes:

All Attributes inherited from parent class

Operations:

EcUtLoggerRelASec - constructor - accepts application name and an optional application version as arguments.
Arguments:appname:EcTChar * appversion:EctChar * = NULL
Return Type:Void
Privilege:Public

SetMyEventType - Used to set the event type.
Arguments:EcTInt
Return Type:EcTVoid
Privilege:Public

~EcUtLoggerRelASec - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcUtLoggerRelASec class has associations with the following classes:
None

4.4.6.3.8 strstream Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
standard c++ class

Attributes:

None

Operations:

None

Associations:

The strstream class has associations with the following classes:
None

4.4.6.4 Dynamic Model Scenarios

4.4.6.4.1 Scenario #1

4.4.6.4.1.1 Abstract

In this scenario, a EcUtLoggerRelA object will be created and will have a multi part message written.

4.4.6.4.1.2 Interfaces

CSS's EcUtLoggerRelA object - for application logging.

4.4.6.4.1.3 Stimulus

This object is used to log event messages to file.

4.4.6.4.1.4 Desired Response

When used, the object should commit the messages to the appropriate file, with 'permission' from the management object.

4.4.6.4.1.5 Participating Classes

EcUtLoggerRelA is the only object involved.

4.4.6.4.1.6 Beginning State

Before being instantiated, the error will not be written in a file.

4.4.6.4.1.7 Ending State

After use, the message that EcUtLoggerRelA was sent will have been written to the appropriate log file.

4.4.6.4.1.8 Scenario Description

In this scenario, the object will be created, have a multi part message written to it, and will be destroyed.

1. The application instantiates a EcUtLoggerRelA object.
2. The application writes a message, with chained <<s assuming the error is not masked out by the management object. (For example if the disk is too full, the management object may decide to block all but the highest level error messages.)
3. Each part of the message is stored in the object, and at each stage the object returns itself so that the <<s can be chained.
4. Finally, the object is passed the ENDMSG marker which causes the file to be opened, locked, checked for its size. Then the message stored internally is written, and finally it is flushed committing all error messages to disk.
5. Then the object is destroyed.

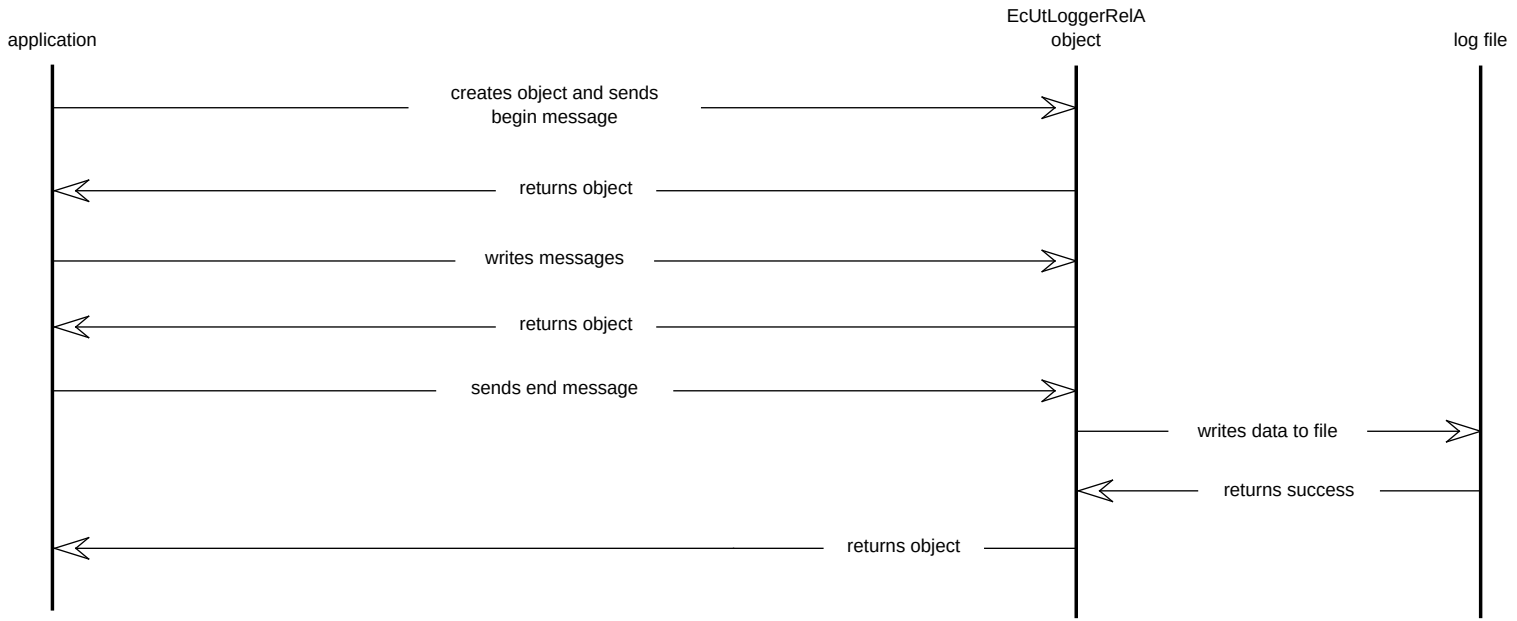

Figure 4.4.6.4-1. Event Logging Service Event Trace

4.4.6.4.2 Scenario #2

4.4.6.4.2.1 Abstract

In this scenario, EcUtLoggerRelAFault object will be created by a management object and will have management fault event written to the log file.

4.4.6.4.2.2 Interfaces

CSS's EcUtLoggerRelAFault object - for management fault event logging

4.4.6.4.2.3 Stimulus

EcUtLoggerRelAFault object is used by a management object to log management event messages to file.

4.4.6.4.2.4 Desired Response

When used, EcUtLoggerRelAFault should commit the messages to the appropriate file.

4.4.6.4.2.5 Participating Classes

Management object

EcUtLoggerRelAFault

4.4.6.4.2.6 Beginning State

Before being instantiated, the error will not be written in a file.

4.4.6.4.2.7 Ending State

After use, the message that EcUtLoggerRelAFault was sent will have been written to the appropriate log file.

4.4.6.4.2.8 Scenario Description

In this scenario, the application will log a management fault event into the specified log file. The error severity is low, SNMP trap will not be generated.

1. The application instantiates a management object.
2. The management object instantiates CSS's EcUtLoggerRelAFault object.
3. The application writes messages.
4. Each part of the message is stored in the EcUtLoggerRelAFault object, and at each stage the object returns itself so that the <<s can be chained.
5. The EcUtLoggerRelAFault object is passed the ENDMSG marker which causes the file to be opened, locked, checked for its size. Then the message stored internally is written, and finally it is flushed committing all error messages to disk.
6. The application check the severity of the event.
7. The severity is low, the application calls the object destructor which frees all memory associated with the object.

4.4.6.4.2.9 Event Trace

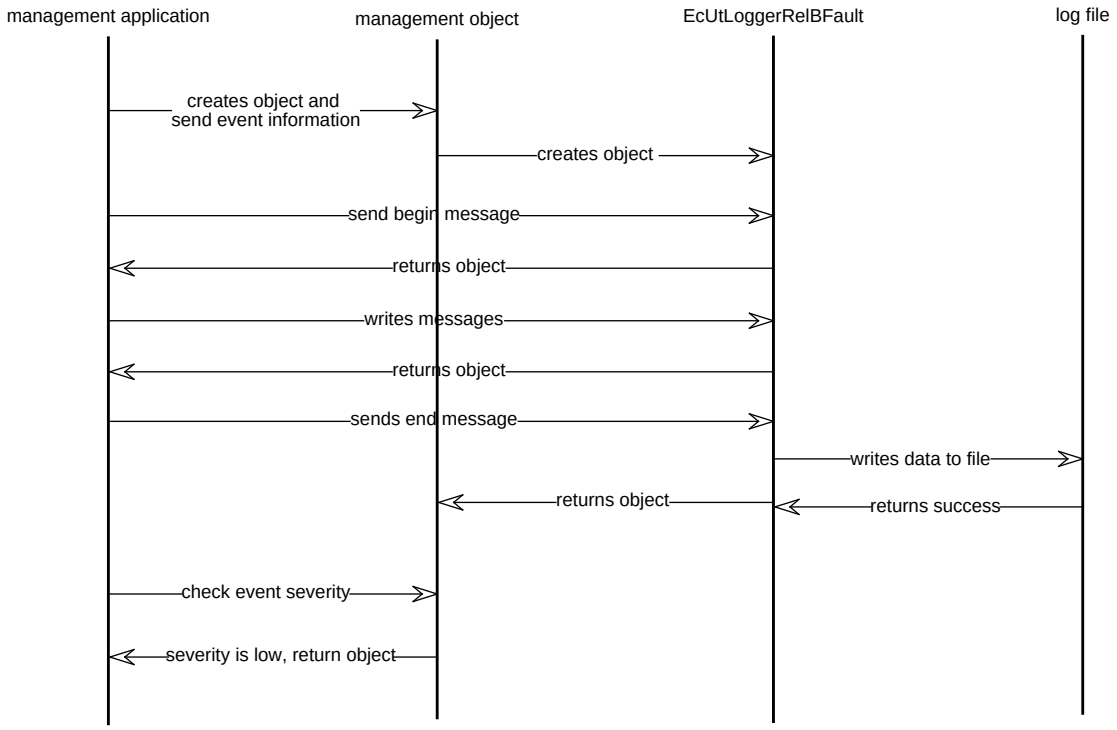

Figure 4.4.6.4-2. Event Logging Service Event Trace for Management Events (Release B)

4.4.6.5 Functional Model and Implementation

The object, EcUtLoggerRelA, will be assigned a unique identifier(eventid) upon instantiation. It will then read a configuration file to determine the name of the file to log into as well as the maximum size for the file log. Messages will be passed in with the << operator. There will be special reserved symbols to end the message and change priority of the messages logged. When a log file is half the maximum size the file will be moved to have a .old extension,. and future writes will occur to a new file. There will be a bitmask to determine what errors will be logged. This can be set by the Management Object. To send an error message a developer would do the following:

loggingobject << BEGINMSG << "This is a error message" << " with multiple strings and the number " << 5 << ENDMSG;

It is necessary end each message with ENDMSG since that will flush all the data and allow other threads to begin writing their own messages. If priority of a message is set, it will be reset with the end of a message. Since beginning to write a message requires an exclusive lock, care should be taken to avoid delay between start and end of a message. The functions will be thread safe. Internally it will use an ofstream so all standard stream data can be sent to the error class. However, even though it uses an strstream internally, only the << constructor and destructor operators are supported. The use of any of the underlying operators is prohibited.

The EcUtLoggerRelAMgmt object is an abstract class with the added fields necessary for logging DCE events. It also has a variable of event type which is set by the classes that inherit from the object. The object will have the ability to determine the severity of the events that it is given and will optionally (as determined by information from the management object) instantiate a M&O object which will create a SNMP trap.

EcUtLoggerRelAFault will set the type to Fault but will otherwise be identical.

EcUtLoggerRelAPerf will set the type to Performance but will otherwise be identical.

EcUtLoggerRelAAudit will set the type to Audit but will otherwise be identical.

EcUtLoggerRelASec will set the type to Security but will otherwise be identical.

The EcUtLoggerRelADebug object will inherit from the EcUtLoggerRelA object and has, in addition, the ability to turn on and off the logging at compile time or runtime.

4.4.6.6 Service/CSCI Management and Strategy

4.4.6.6.1 System Management and Strategy

The table to determine the name of the log file and the maximum file size will be maintained by M&O. The management object will also be able to selectively enable or disable error logging to protect against filling a disk.

4.4.6.6.2 Operator Interface

There is no direct Operator Interface.

4.4.6.6.3 Reports

No reports are sent externally, but there is data logged locally, and the possibility that traps can be sent.

4.4.7 Bulk Data Service (BDS)

ECS Release B will use a lightweight TCP/IP protocol, called Bulk Data Service (BDS), for file transfer between Data Server and Science Processing. BDS provides an NFS-like client interface, and its server is supported as a thread in the operating system kernel. BDS has been demonstrated to support file transfer over a single HiPPI channel between SGI platforms at a rate of 65 MB per second; three channels running simultaneously between two machines have supported 180 MB per second. This speed is accomplished by providing separate socket connections for control and data between the client and server, by using the largest block sizes supported by the underlying fabric, by using direct I/O to the client and host file systems (avoiding buffering by the operating system), and by aligning data buffers with memory buffer boundaries to allow efficient memory transfers. BDS uses TCP/IP services and is independent of NFS and independent of the physical network. BDS is available from SGI as a beta product in March 1996, and will be available as an Irix 6.2 extension product in June 1996.

4.5 Infrastructure Services

4.5.1 Process Framework

4.5.1.1 Overview

The ECS contains several infrastructure features which facilitate the implementation of client-server applications. The framework provides an extensible mechanism for ECS Client and Server applications to transparently include these infrastructure features. Therefore, its importance grows with future releases of ECS. Furthermore, the framework is used solely by ECS custom developed applications and as such is not meant for COTS applications. The primary objective of the PF is to ensure design and implementation consistency for all ECS Client and Server applications. This is achieved by encapsulating the implementation details of ECS infrastructure services and removing the need for programmers to rewrite common initialization code.

In general, the following capabilities are needed in the ECS client and server applications and have to be accommodated in a appropriate fashion:

1) Ability to initialize the process application and infrastructure in a consistent way and provide some basic process information
2) Interface to Mode Management
3) Interface to Error-Event logging
4) Ability to set OODCE Naming/Directory Service options
5) Ability to set OODCE Security management parameters
6) Support for Life Cycle services
7) Interface to Asynchronous Message Passing
8) Interface to Server Request Framework, and
9) Interface to the Batch FTP service

A two step approach is used to develop the process framework. First, a process classification for the ECS project is developed from the client/server perspective. Then, the required capabilities are allocated at different levels of abstraction for each process type. The details of the above steps are presented below.

Process Classification:

Figure 4.5.1.1-1 presents a classification of ECS application processes. A generic process can be specialized into a client process and a server process. The former can be specialized into a gateway client (client external to the ECS system and connected to it through a gateway) and DCE client (client in the ECS system which uses DCE communication mechanisms). A server process can be specialized into an unmanaged server process and a managed server process.

In this context, a Managed Server Process is a process controlled by the management agent services provided by MSS. An Unmanaged server process is a process created by another process for its specific use and is not under the direct control of the management agents. The name must not confuse the reader. An unmanaged process is managed as well, but it is managed by its parent

process instead of MSS. Finally, it should be noted that a process, which is both client and server, is considered a server.

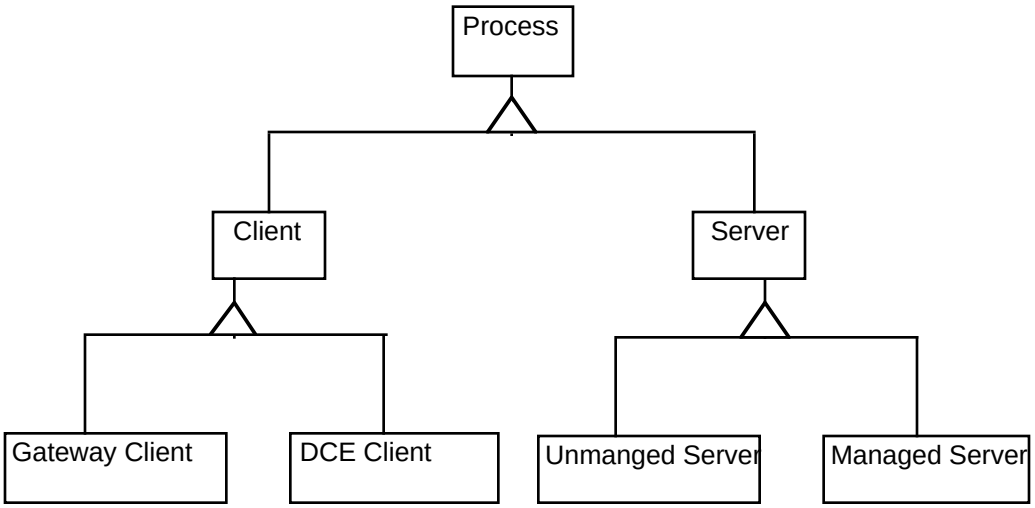

Figure 4.5.1.1-1. ECS Process Classification

Capability Allocation:

All the capabilities that are common to all the processes need to be incorporated at the generic process level in the hierarchy described above. These include, the basic process information capabilities, and interfaces to Mode Management and Event Logging. The ability to locate services and set security preferences is provided to the client processes and server process in different ways. Therefore, this functionality is implemented differently in the client and server but the interface is common.

General services such as asynchronous messaging, Server Request Framework access, and batch FTP that are needed by all server processes are provided by interfacing with the Message Passing Service, the Server Request Framework and the batch FTP facility respectively. Full Life cycle functionality is provided to every Server process. It has to be tailored in a different way for Managed or Un-Managed Servers. The former has to be implemented with the management instrumentation classes being developed by the Management Subsystem (MSS). The latter has to be implemented by the parent Server which spawned the un-managed server.

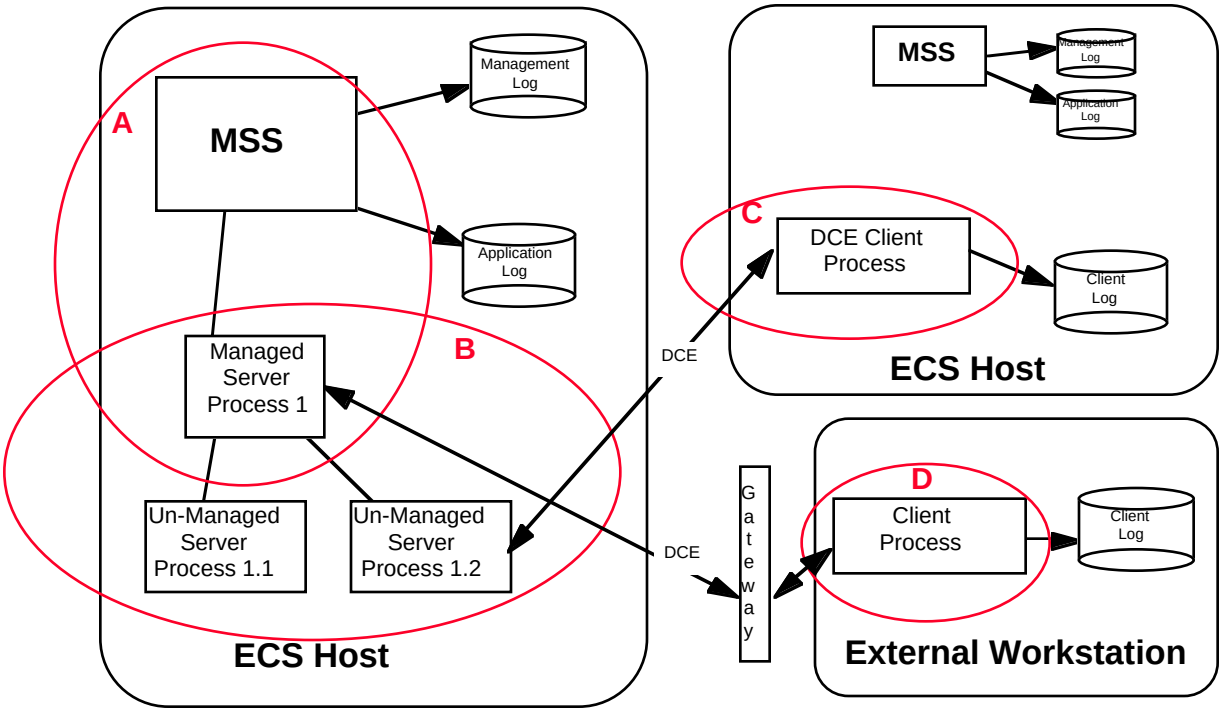

Figure 4.5.1.2-1. Process Framework Context

4.5.1.2 Context

Figure 4.5.1.2-1 identifies the context in which the above described processes operate. The ellipse A, B, C, and D locate the contexts for Managed Server Processes, Unmanaged Server Processes, DCE Client Processes, and Gateway Client Processes. A Managed Server process is connected to the MSS sub-system (ellipse A). It can also spawn other processes. These latter processes might be other Managed Server Processes as well as Unmanaged Server Processes. Unmanaged Server Processes are connected to the parent process which has spawned them (ellipse B). They are not directly connected to MSS and MSS is not aware of their existence. Their world is limited to the application domain. They are called unmanaged because MSS doesn't manage them directly. Client Processes have less constraints. They are simply clients of ECS server applications with an external interface to log information in the *Client Log*. Only the DCE clients (ellipse C) are part of the ECS and for only these kind of client processes the framework is applicable.

4.5.1.3 Object Model

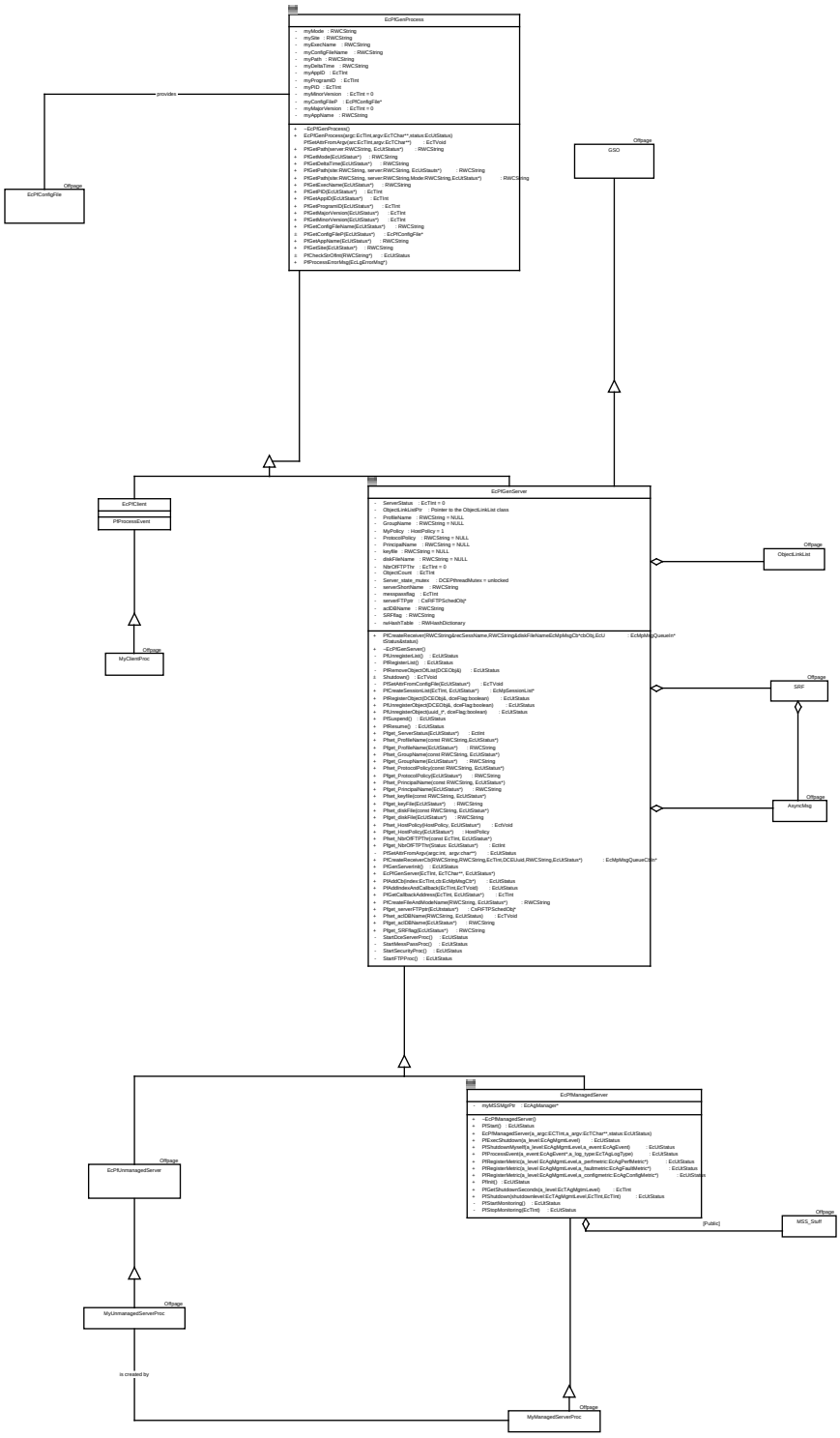

Figure 4.5.1.3-1. Process Framework Object Model

4.5.1.3.1 AsyncMsg Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The AsyncMsg class has associations with the following classes:
EcPfGenServer (Aggregation)
SRF (Aggregation)

4.5.1.3.2 EcPfClient Class

Parent Class:EcPfGenProcess

Attributes:

All Attributes inherited from parent class

Operations:

PfProcessEvent
Arguments:

Associations:

The EcPfClient class has associations with the following classes:
None

4.5.1.3.3 EcPfConfigFile Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The EcPfConfigFile class has associations with the following classes:
Class: EcPfGenProcess provides

4.5.1.3.4 EcPfGenProcess Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
EcPfGenProcess represents the process framework for a generic process. It has all the common functionality for all the process frameworks. It is mainly a container of attributes needed by every process. It obtains attribute values from the configuration file or the command line parameters.

Attributes:

myAppID - Application ID.
Data Type:EcTInt
Privilege:Private
Default Value:

myAppName - Name of the application
Data Type:RWCString
Privilege:Private
Default Value:

myConfigFileName - Full configuration file name path. It must be provided in the command line.
Data Type:RWCString
Privilege:Private
Default Value:

myConfigFileP - A pointer to the configuration file.
Data Type:EcPfConfigFile*
Privilege:Private
Default Value:

myDeltaTime - Delta time for simulation purposes
Data Type:RWCString
Privilege:Private
Default Value:

myExecName - Executable name of the program
Data Type:RWCString
Privilege:Private
Default Value:

myMajorVersion - Major version of the application
Data Type:EcTInt
Privilege:Private
Default Value:0

myMinorVersion - Minor version of the application.
Data Type:EcTInt
Privilege:Private
Default Value:0

myMode - Provides the mode the user is curently in (i.e. test, production, training, etc...). It must be privided from the command line.
Data Type:RWCString
Privilege:Private
Default Value:

myPID - Process ID
Data Type:EcTInt
Privilege:Private
Default Value:

myPath - CDS entry path.
Data Type:RWCString
Privilege:Private
Default Value:

myProgramID - Program ID.
Data Type:EcTInt
Privilege:Private
Default Value:

mySite - Site name where the process is running.
Data Type:RWCString
Privilege:Private

Default Value:

Operations:

EcPfGenProcess - Constructor which will set attributes from configuration file. It will then read arguments from the command line to reset attributes dynamically.
Arguments:argc:EcTInt,argv:EcTChar**,status:EcUtStatus
Return Type:Void
Privilege:Public

PfCheckStrOfInt - Check input data type for integer
Arguments:RWCString*
Return Type:EcUtStatus
Privilege:Protected

PfGetAppID - Obtains the application ID
Arguments:EcUtStatus*
Return Type:EcTInt
Privilege:Public

PfGetAppName - Obtains application name
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetConfigFileName - Obtains the full path name of the configuration file
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetConfigFileP - Obtains the pointer to the configuration file
Arguments:EcUtStatus*
Return Type:EcPfConfigFile*
Privilege:Protected

PfGetDeltaTime - Obtains delta time
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetExecName - Obtains the executable name
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetMajorVersion - Obtains the major version of the program
Arguments:EcUtStatus*
Return Type:EcTInt
Privilege:Public

PfGetMinorVersion - Obtains the minor version of the program
Arguments:EcUtStatus*
Return Type:EcTInt
Privilege:Public

PfGetMode - Obtains the mode
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetPID - Obtains the process ID of the program
Arguments:EcUtStatus*
Return Type:EcTInt
Privilege:Public

PfGetPath - Construct the CDS path entry using mySite, myMode and the specified server name
Arguments:server:RWCString, EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetPath - Construct the CDS path entry using myMode, and the specified site name and server name
Arguments:site:RWCString, server:RWCString, EcUtStauts*
Return Type:RWCString
Privilege:Public

PfGetPath - Construct the CDS path entry using the specified Mode, site name and server name
Arguments:site:RWCString, server:RWCString,Mode:RWCString,EcUtStatus*
Return Type:RWCString
Privilege:Public

PfGetProgramID - Obtains the program ID
Arguments:EcUtStatus*
Return Type:EcTInt
Privilege:Public

PfGetSite - Obtains the site name

Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

PfProcessErrorMsg - TBD
Arguments:EcLgErrorMsg*
Return Type:Void
Privilege:Public

PfSetAttrFromArgv - Read argument list from command line and set the attributes
Arguments:arc:EcTInt,argv:EcTChar**
Return Type:EcTVoid
Privilege:Protect

~EcPfGenProcess - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcPfGenProcess class has associations with the following classes:
Class: EcPfConfigFile provides

4.5.1.3.5 EcPfGenServer Class

Parent Class:EcPfGenProcess
Parent Class:GSO
Public:No
Distributed Object:No
Purpose and Description:
This class inherits from both the EcPfGenProcess and the DCEServer class and is inherited by the EcPfManagedServer class. This class will provide the generic DCEServer capabilities and some enhanced features as well. It will also serve as a linking interface with other services such as FTP, Message Passing, Server Request Framework, and Security.

Attributes:

GroupName - Needed to decide the group name in CDS
Data Type:RWCString
Privilege:Private
Default Value:NULL

MyPolicy - Decides the host policy; 0 for unique host policy, 1 for multiple host policy.
Data Type:HostPolicy
Privilege:Private
Default Value:1

NbrOfFTPThr - Indicates the number of FTP threads in the application; The FTP initialization is done only for values greater then 0.
Data Type:EcTInt
Privilege:Private
Default Value:0

ObjectCount - Keeps track of the number of objects in the link list
Data Type:EcTInt
Privilege:Private
Default Value:

ObjectLinkListPtr - Pointer to the linked list of objects to be registered or unregistered.
Data Type:Pointer to the ObjectLinkList class
Privilege:Private
Default Value:

PrincipalName - Needed in message passing and security, user name
Data Type:RWCString
Privilege:Private
Default Value:NULL

ProfileName - Needed to decide the profile name in CDS
Data Type:RWCString
Privilege:Private
Default Value:NULL

ProtocolPolicy - Needed to decide the protocol policy of CDS
Data Type:RWCString
Privilege:Private
Default Value:NULL

SRFflag - Flag indicating whether Server Request Framework is needed
Data Type:RWCString
Privilege:Private
Default Value:

ServerStatus - There is to indication the status of server objects. There are three different status: 1) Initial - no user objects registered with the server, 2) suspendable and 3) resumable
Data Type:EcTInt

Privilege:Private
Default Value:0

Server_state_mutex - Lock attribute used to activate a self unlocking mutex that forces register object, unregister object, suspend, resume, to be called sequentially.
Data Type:DCEPthreadMutex
Privilege:Private
Default Value:unlocked

aclDBName - Filename for ACL DB
Data Type:RWCString
Privilege:Private
Default Value:

diskFileName - Needed as a parameter in message passing
Data Type:RWCString
Privilege:Private
Default Value:NULL

keyfile - Needed as a parameter in message passing and security
Data Type:RWCString
Privilege:Private
Default Value:NULL

messpassflag - Flag that is put to 1 by the user in order to explicitly require that message passing be activated.
Data Type:EcTInt
Privilege:Private
Default Value:

rwHashTable - Contains FTP thread index and associated callback functions
Data Type:RWHashDictionary
Privilege:Private
Default Value:

serverFTPptr - Pointer to FTP object
Data Type:CsFtFTPSchedObj*
Privilege:Private
Default Value:

serverShortName - server name
Data Type:RWCString
Privilege:Private
Default Value:

Operations:

EcPfGenServer - EcPfGenServer Constructor
Arguments:EcTInt, EcTChar**, EcUtStatus*
Return Type:Void
Privilege:Public

PfAddCb - This will calls the respective message passing method. (For details see description of EcMpMsgPsngCtrl class)
Arguments:index:EcTInt,cb:EcMpMsgCb*
Return Type:EcUtStatus
Privilege:Public

PfAddIndexAndCallback - This method is called by the application programmer any time they make a FTP call. Its purpose is to keep the callback together with its index in a HashTable, so that in case of failure, given the index, the address of the callback can be traced.
Arguments:EcTInt,EcTVoid
Return Type:EcUtStatus
Privilege:Public

PfCreateFileAndModeName - This method is used to create filenames with the extension _mode, so that be explicitly forced that when applications are run in different modes, the files used for Security, Message Passing, etc, be different.
Arguments:RWCString, EcUtStatus*
Return Type:RWCString
Privilege:Public

PfCreateReceiver - Method for creating a receiver object for message passing
Arguments:
RWCString&recSessName,RWCString&diskFileNameEcMpMsgCb*cbObj,EcUtStatus&status
Return Type:EcMpMsgQueueIn*
Privilege:Public

PfCreateReceiverCb - Method for creating a receiver object for message passing
Arguments:RWCString,RWCString,EcTInt,DCEUuid,RWCString,EcUtStatus*
Return Type:EcMpMsgQueueCbIn*
Privilege:Public

PfCreateSessionList - Creates session list for message passing
Arguments:EcTInt, EcUtStatus*
Return Type:EcMpSessionList*
Privilege:Public

PfGenServerInit - Starts the main attributes of the server obtained in configuration file and command line
Arguments:
Return Type:EcUtStatus
Privilege:Public

PfGetCallbackAddress - This method is called by FTP service in case of failure. Given the index, it will trace the address of the callback.
Arguments:EcTInt, EcUtStatus*
Return Type:EcTInt
Privilege:Public

PfRegisterList - This method is called by PfResume for re-registering objects with the server
Arguments:
Return Type:EcUtStatus
Privilege:Private

PfRegisterObject - Method for registering objects with the server and inserting them in a private list
Arguments:DCEObj&, dceFlag:boolean
Return Type:EcUtStatus
Privilege:Public

PfRemoveObjectOfList - This method is called by PfUnregisterObject for removing an objects from the linked list
Arguments:DCEObj&
Return Type:EcUtStatus
Privilege:Private

PfResume - Method for re-registering the objects that reside in the linked list
Arguments:
Return Type:EcUtStatus
Privilege:Public

PfSetAttrFromArgv - This method is called by the constructor to check for any possible attributes that need to be set. It will overwrite the attribute values already read in the configuration file.
Arguments:argc:int, argv:char**
Return Type:EcUtStatus
Privilege:Private

PfSetAttrFromConfigFile - Read the configuration file and sitting class attribuites accordingly

Arguments:EcUtStatus*
Return Type:EcTVoid
Privilege:Private

PfSuspend - Method for unregistering all objects that are inserted in the linked list
Arguments:
Return Type:EcUtStatus
Privilege:Public

PfUnregisterList - This method is called by PfSuspend for unregistering objects from the server
Arguments:
Return Type:EcUtStatus
Privilege:Private

PfUnregisterObject - Method for unregistering a given object with the server and removing it from the private list
Arguments:DCEObj&, dceFlag:boolean
Return Type:EcUtStatus
Privilege:Public

PfUnregisterObject - Method for unregistering a given object with the server and removing it from the private list
Arguments:uuid_t*, dceFlag:boolean
Return Type:EcUtStatus
Privilege:Public

Pfget_GroupName - Returns GroupName value
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_HostPolicy - Return HostPolicy value
Arguments:EcUtStatus*
Return Type:HostPolicy
Privilege:Public

Pfget_NbrOfFTPThr - Returns NbrOfFTPThr value
Arguments:Status: EcUtStatus*
Return Type:EctInt
Privilege:Public

Pfget_PrincipalName - Returns PrincipalName value
Arguments:EcUtStatus*
Return Type:RWCString

Privilege:Public

Pfget_ProfileName - Returns the ProfileName value
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_ProtocolPolicy - Returns the ProtocolPolicy value
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_SRFflag - Return Server Request Framework flag
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_ServerStatus - Returns server status _ServerStatus
Arguments:EcUtStatus*
Return Type:EctInt
Privilege:Public

Pfget_aclDBName - Returns file name for ACL database
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_diskFile - Returns diskFile value
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_keyFile - Returns keyfile value
Arguments:EcUtStatus*
Return Type:RWCString
Privilege:Public

Pfget_serverFTPptr - Returns pointer to FTP object
Arguments:EcUtstatus*
Return Type:CsFtFTPSchedObj*
Privilege:Public

Pfset_GroupName - Set group name
Arguments:const RWCString, EcUtStatus*
Return Type:Void

Privilege:Public

Pfset_HostPolicy - Set HostPolicy value
Arguments:HostPolicy, EcUtStatus*
Return Type:EctVoid
Privilege:Public

Pfset_NbrOfFTPThr - Set NbrOfFTPThr value
Arguments:const EcTInt, EcUtStatus*
Return Type:Void
Privilege:Public

Pfset_PrincipalName - Set PrincipalName value
Arguments:const RWCString, EcUtStatus*
Return Type:Void
Privilege:Public

Pfset_ProfileName - set profile name value
Arguments:const RWCString,EcUtStatus*
Return Type:Void
Privilege:Public

Pfset_ProtocolPolicy - Set ProtocolPolicy value
Arguments:const RWCString, EcUtStatus*
Return Type:Void
Privilege:Public

Pfset_aclDBName - Sets the ACL DB file name attribute
Arguments:RWCString, EcUtStatus
Return Type:EcTVoid
Privilege:Public

Pfset_diskFile - Set diskFile value
Arguments:const RWCString, EcUtStatus*
Return Type:Void
Privilege:Public

Pfset_keyfile - Set keytab file location
Arguments:const RWCString, EcUtStatus*
Return Type:Void
Privilege:Public

Shutdown - This is theServer Shutdown method
Arguments:
Return Type:EcTVoid

Privilege:Protected

StartDceServerProc - This method is called privately by the PfGenStart to initialize common server activities such as SetName, SetProtocols, etc. It will only perform those activities for which the respective attributes will be set either in the configuration file or in the command line.
Arguments:
Return Type:EcUtStatus
Privilege:Private

StartFTPProc - This method is called privately by the PfGenStart to initialize the FTP process when the application sets the number of FTP threads to a value greater than 0. The maximum number of FTP threads allowed is 30.
Arguments:
Return Type:EcUtStatus
Privilege:Private

StartMessPassProc - This method is called privately by the PfGenStart to initialize the message passing process.
Arguments:
Return Type:EcUtStatus
Privilege:Private

StartSecurityProc - This method is called by PfGenServerInit to initialize the Security process.
Arguments:
Return Type:EcUtStatus
Privilege:Private

~EcPfGenServer - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcPfGenServer class has associations with the following classes:
None

4.5.1.3.6 EcPfManagedServer Class

Parent Class:EcPfGenServer
Public:Yes
Distributed Object:No
Purpose and Description:

This class is the framework class for Managed Server Processes. This class defines the method Process Event which handles the events generated by the Managed Server Processes. This class is also connected to the MSS EcAgManager class as required by the MSS desing. The Managed Server class will povide methods to inform the EcAgManager to start and stop monitoring, to inform the EcAgManager the number of shutdown seconds required for the application, program, and process, to register metrics with the EcAgManager. The Managed Server class also receives requests from the EcAgManager class to suspend, resume, and shutdown. The method, PfShutdownMyself, is provided by the Managed Server class to the application may request a shutdown of itself.

Attributes:

myMSSMgrPtr - Pointer to the EcAgManger object. Used to call methods contained in the EcAgManager class.
Data Type:EcAgManager*
Privilege:Private
Default Value:

Operations:

EcPfManagedServer - Constructor
Arguments:a_argc:ECTInt,a_argv:EcTChar**,status:EcUtStatus
Return Type:Void
Privilege:Public

PfExecShutdown - Calls theServer Shutdown() and PfShutdown method with the management level(application, program, process).
Arguments:a_level:EcAgMgmtLevel
Return Type:EcUtStatus
Privilege:Public

PfGetShutdownSeconds - Needs to be overriden by application, the number of seconds requires to shutdown the application
Arguments:a_level:EcTAgMgtmLevel
Return Type:EcTInt
Privilege:Public

PfInit - Calls the PfGenServerInit(inherited) method, instantiates the EcAgManager Object and registers it with the GSO
Arguments:
Return Type:EcUtStatus
Privilege:Public

PfProcessEvent - Calls the ProcessEvent method of the EcAgManager object

Arguments:a_event:EcAgEvent*,a_log_type:EcTAgLogType
Return Type:EcUtStatus
Privilege:Public

PfRegisterMetric - Calls the RegisterMetric method of the EcAgManger object.
Arguments:a_level EcAgMgmtLevel,a_perfmetric:EcAgPerfMetric*
Return Type:EcUtStatus
Privilege:Public

PfRegisterMetric - Calls the RegisterMetric method of the EcAgManager object.
Arguments:a_level EcAgMgmtLevel,a_faultmetric:EcAgFaultMetric*
Return Type:EcUtStatus
Privilege:Public

PfRegisterMetric - Calls the RegisterMetric method of the EcAgManager object.
Arguments:a_level:EcAgMgmtLevel,a_configmetric:EcAgConfigMetric*
Return Type:EcUtStatus
Privilege:Public

PfShutdown - Needs to be overriden by application to perform specific shutdown appropriate for application
Arguments:shutdownlevel:EcTAgMgmtLevel,EcTInt,EcTInt
Return Type:EcUtStatus
Privilege:Public

PfShutdownMyself - Calls StopMonitoring. Calls PfShutdown method with the management level(application,program,process). Calls the DCEServer Shutdown method.
Arguments:a_level:EcAgMgmtLevel,a_event:EcAgEvent
Return Type:EcUtStatus
Privilege:Public

PfStart - Calls PfStartMonitoring and DCEServer listen methods
Arguments:
Return Type:EcUtStatus
Privilege:Public

PfStartMonitoring - Calls StartMonitoring() method of the EcAgManager Object
Arguments:
Return Type:EcUtStatus
Privilege:Private

PfStopMonitoring - Calls StopMonitoring() method of the EcAgManager Object
Arguments:EcTInt
Return Type:EcUtStatus
Privilege:Private

~EcPfManagedServer - Destructor. Deletes the EcAgManager object.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcPfManagedServer class has associations with the following classes:
None

4.5.1.3.7 EcPfUnmanagedServer Class

Parent Class:EcPfGenServer

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The EcPfUnmanagedServer class has associations with the following classes:
None

4.5.1.3.8 GSO Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The GSO class has associations with the following classes:
None

4.5.1.3.9 MSS_Stuff Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The MSS_Stuff class has associations with the following classes:
EcPfManagedServer (Aggregation)

4.5.1.3.10 MyClientProc Class

Parent Class:EcPfClient

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The MyClientProc class has associations with the following classes:
None

4.5.1.3.11 MyManagedServerProc Class

Parent Class:EcPfManagedServer

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The MyManagedServerProc class has associations with the following classes:
Class: MyUnmanagedServerProc iscreatedby

4.5.1.3.12 MyUnmanagedServerProc Class

Parent Class:EcPfUnmanagedServer

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The MyUnmanagedServerProc class has associations with the following classes:
Class: MyManagedServerProc iscreatedby

4.5.1.3.13 ObjectLinkList Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

None

Operations:

None

Associations:

The ObjectLinkList class has associations with the following classes:
EcPfGenServer (Aggregation)

4.5.1.3.14 SRF Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The SRF class has associations with the following classes:
EcPfGenServer (Aggregation)

4.5.1.4 Functional Model

Different types of processes need different framework functionality. The framework design closely maps the process classification so that the reuse of common functionality is maximized.

The class `EcPfGenProcess` represents the process framework for a generic process. It has all the common functionality for all the processes. It is an abstract class which provides a common interface to access the common methods. For example, every process has to have a mode management capability. Moreover, every process needs to have available some basic information about itself. This functionality can work in the same way for every process and therefore is defined in this class. On the other hand, every process needs to have Event-Error Handling capability, but its implementation must be different for client and server processes, although its interface could be the same. The method *`PfProcessEvent`* is therefore declared as pure virtual forcing the subclasses to implement this capability.

The class `EcPfClient` defines the framework for client processes. The only method specified is the *`PfProcessEvent`* method. The current design assumes that the client process does not differ from a generic process in any other respect. This class is defined to satisfy a future need for specific functionality needed by client processes.

The class `EcPfGenServer` is the basic framework class for server process. It is further specialized into Managed and Un-Managed Server Process. All the common functionality for every Server is included in this class.

The functionality includes:

- Lifecycle support, through the use of the methods: *PfShutdown*, *PfSuspend*, and *PfResume*.
- The Message Passing interface through the methods: *PfCreateSessionList*, *PfCreateReceiver*, and *PfCreateReceiverCB.*
- The Server Request Framework interface, through the pointer to the SRF Handler object.
- Ability to set OODCE parameters. Several access functions (Get and Set methods) are provided to read and change the attribute values.
- The customization of a select set of GSO methods that are commonly used by the application developers.

The class EcPfManagedServer is the framework class for Managed Server Processes. It defines the method *PfProcessEvent* which handles the events generated by Managed Server Processes. It is also connected to the MSS classes as required by the MSS design. The class EcPfUnmanagedServer is the framework class for Unmanaged Server Processes. The blocks *myClientProc*, *myUnmanagedServerProc*, and *MyManagedServerProc* represent the possible classes developed for each specific applications by the developer.

Table 4.5.1.4-1 describes the PF classes.

Table 4.5.1.4-1. PF Object Responsibility Matrix

Class name	Description
EcPfGenProcess	represents the process framework for a generic process. It has all the common functionality for all the processes. It is an abstract class which provides a common interface to access the common methods.
EcPfClient	defines the framework for client processes.
EcPfGenServer	basic framework class for server process.
EcPfManagedServer	is the framework class for Managed Server Processes.
EcPfUnManagedServer	is the framework class for Unmanaged Server Processes.
MyManagedServerProc	is the class implemented by the programmer to inherit managed server process functionality
MyUnManagedServerProc	is the class implemented by the programmer to inherit unmanaged server process functionality
MyClientProc	is the class implemented by the programmer to inherit client process functionality

4.5.1.5 Dynamic Model Scenarios

4.5.1.5.1 Scenario #1

4.5.1.5.1.1 Abstract

This scenario describes the startup of a generic server application.

4.5.1.5.1.1 Interfaces

MSS provided classes.

4.5.1.5.1.2 Stimulus

The operator using HP/Open View (HPOV) double clicks on the server application icons.

4.5.1.5.1.3 Desired Response

The application is started and an icon representing the running application appears on the HPOV desktop.

4.5.1.5.1.4 Participating Classes

MyManagedServerProc

EcPfManagedServer

GSO

SRF

EcMpSessionList

EcPfConfigFile

EcAgManager

We also consider in the event trace the MSS SubAgent block and the application main program module to better clarify the system behavior.

4.5.1.5.1.5 Pre-conditions

The application is available on an ECS host.

4.5.1.5.1.6 Post-conditions

The application is started.

4.5.1.5.1.7 Scenario Description

The request to start an ECS application is issued by a ECS systems administrator who has a HP/ Open View based user interface which reaches the MSS sub-agent. The subagent then fires the associated script needed to start the ECS server application. The script in turn runs the server main program. Specific information for ECS applications is stored in configuration files. These include the attributes needed for the process execution thus reducing the number of parameters to be passed to the main program and minimizing the need for environment variables.

Application main first instantiates a `MyManagedServerProc` object, and reads from a configuration file and command line arguments to set attributes needed for the process execution. Application main then calls `PfInit()` to perform all DCE related initialization (and optionally SRF, message passing, and FTP initialization) for the process, and creates an `EcAgManager` object. `EcAgManager` is a distributed object which provides interfaces to the MSS sub-agent. The application will then perform normal server set up such as creating server manager objects, registering objects with GSO, and registering metrics with MSS. Finally, application main will call `PfStart()` to start MSS monitoring, and listen to client requests.

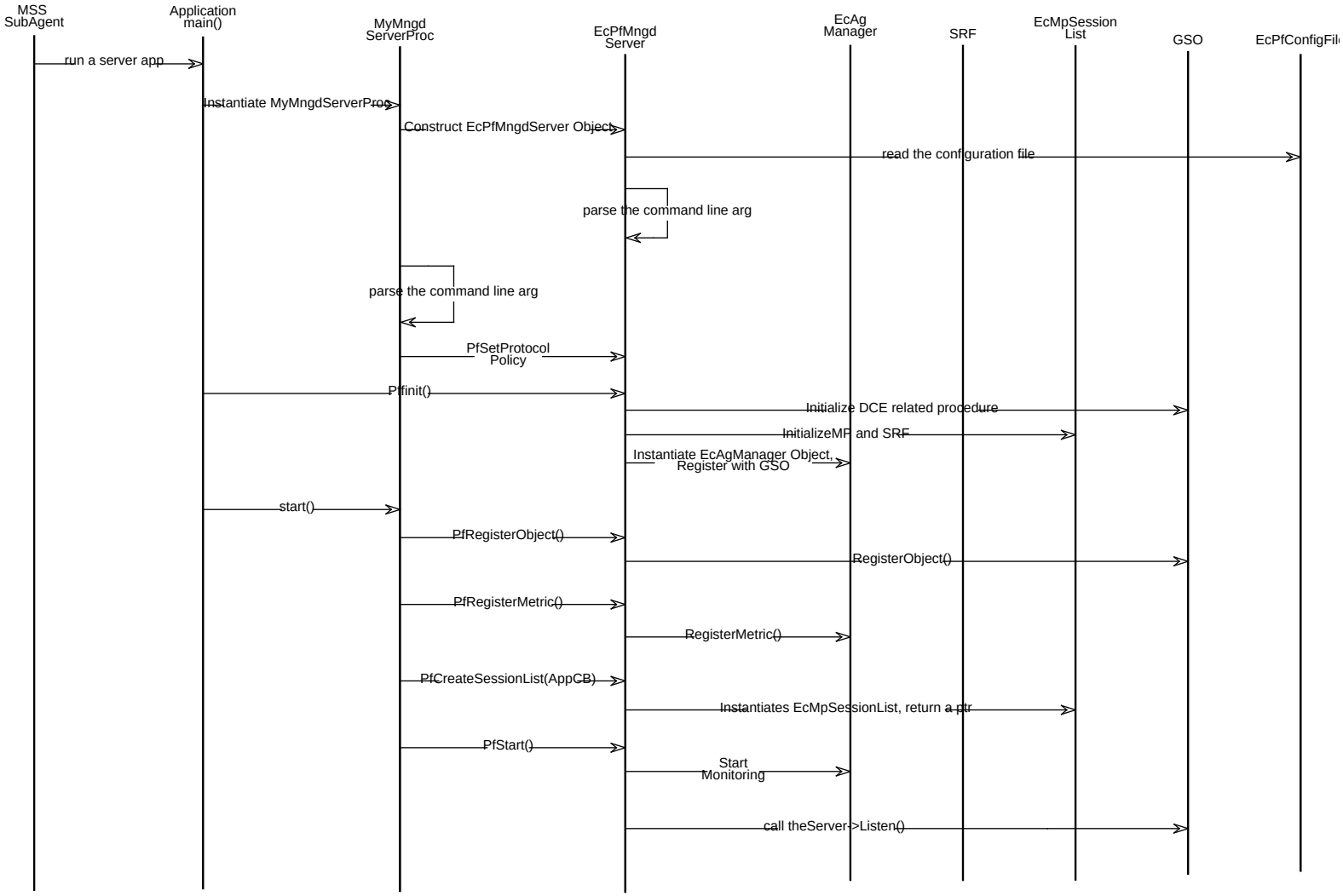

Figure 4.5.1.5-1 Process Framework Event Trace for Application Start-Up

4.5.1.5.2 Scenario #2

4.5.1.5.2.1 Abstract

This scenario describes the shutdown of a generic server application at the request of MSS.

4.5.1.5.2.2 Interfaces

MSS

4.5.1.5.2.3 Stimulus

The operator, using HPOV, clicks on the server application icon and selects Shutdown from the appearing menu.

4.5.1.5.2.4 Desired Response

The application is shutdown and its icon, representing the running application, is removed from the HPOV desktop.

4.5.1.5.2.5 Participating Classes

MyManagedServerProc

EcPfManagedServer

GSO

EcAgManager

We also consider in the event trace the MSS SubAgent block and the application main program module to better clarify the system behavior.

4.5.1.5.2.6 Pre-conditions

The application process is running.

4.5.1.5.2.7 Post-conditions

The application process is shutdown.

4.5.1.5.2.8 Scenario Description

MSS sub-agent, `EcAgManager`, calls `PfGetShutdownSec()` to obtain the number of seconds the application needs to perform shutdown. This method needs to be over ridden by the application to provide an estimation of shutdown time the application needs. If `PfGetShutdownSec()` is not provided by the application, a default number of seconds will be provided by the process framework. After the shutdown seconds is obtained, the MSS sub-agent then calls `PfExecShutdown()` to shutdown the server process. `PfExecShutdown()` will first call `theServer->Shutdown()` to stop accepting client requests, and return from listen. The application specific shutdown method (`PfShutdown()`) will then be called to perform application specific shutdown. If the application does not shutdown within the indicated time frame, the operator will be given an opportunity to kill the application or let it continue to operate.

4.5.1.5.2.9 Event Trace

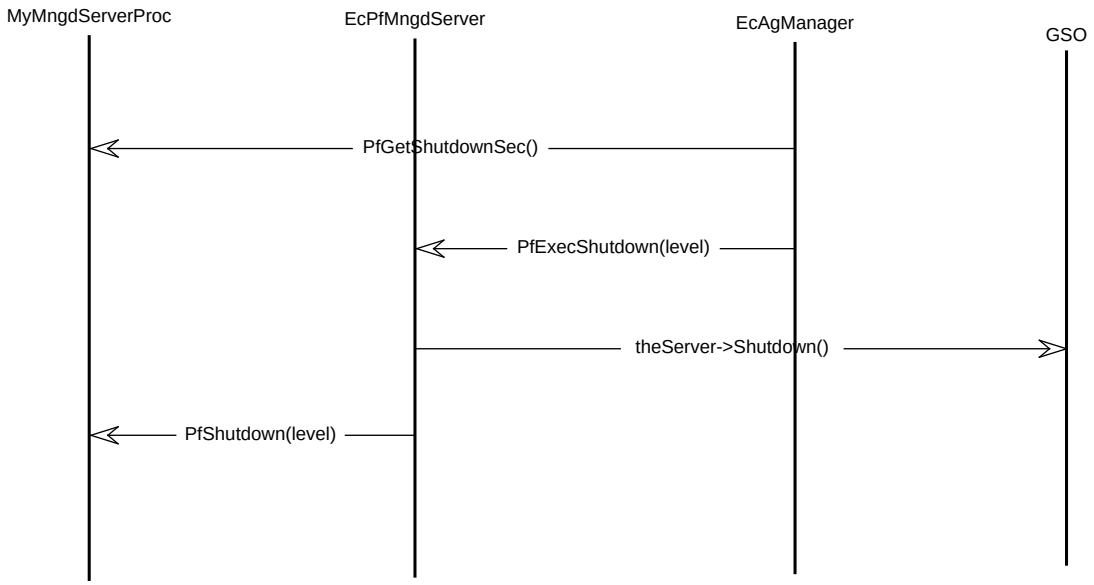

Figure 4.5.1.5-2 Process Framework Event Trace for Application Shutdown by MSS

4.5.1.5.3 Scenario #3

4.5.1.5.3.1 Abstract

The process decides to terminate itself.

4.5.1.5.3.2 Interfaces

None.

4.5.1.5.3.3 Stimulus

An error/exception occurs and the process decides to terminate itself.

4.5.1.5.3.4 Desired Response

The application process is shutdown.

4.5.1.5.3.5 Participating Classes

MyManagedServerProc

EcPfManagedServer

GSO

EcAgManager

4.5.1.5.3.6 Pre-conditions

The application process is running, an error/exception occur within the application.

4.5.1.5.3.7 Post-conditions

The application process is shutdown.

4.5.1.5.3.8 Scenario Description

The application call `PfShutdownMySelf` method, which will first call `PfGetShutdownSec()` to obtain the number of seconds to shutdown. `PfStopMonitoring()` will then be called to notify MSS to stop monitoring the process. A call to `theServer->Shutdown()` is made to shutdown the server and return from listening. Finally, the application shutdown `PfShutdown()` will be called to gracefully shutdown the application. If the application does not shutdown within the indicated time frame, the operator will be given an opportunity to kill the application or let it continue to operate.

4.5.1.5.3.9 Event Trace

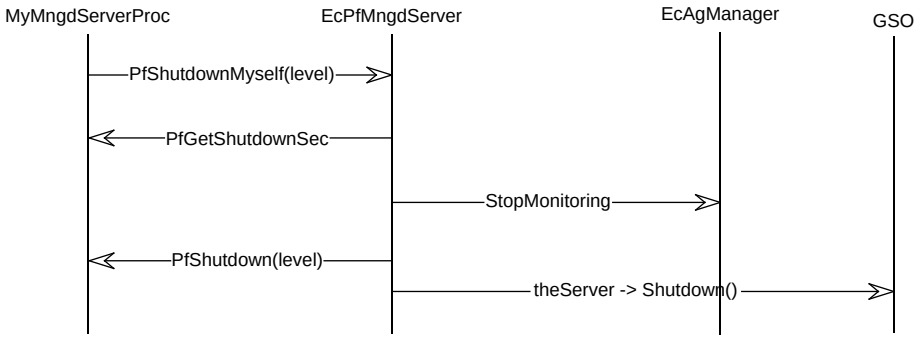

Figure 4.5.1.5-3 Process Framework Event Trace for Application Self Shutdown

4.5.1.5.4 Scenario #4

4.5.1.5.4.1 Abstract

This scenario describes how a generic server application is suspended and then resumed.

4.5.1.5.4.2 Interfaces

MSS provided classes.

4.5.1.5.4.3 Stimulus

The operator using HPOV clicks on the server application process icon and selects Suspend from the appearing menu. Later he selects Resume.

4.5.1.5.4.4 Desired Response

The application is suspended and its icon, representing the running application on the HPOV desktop, changes color. When Resume is selected, the process resume listening for client requests and the HPOV icon go back to the original color.

4.5.1.5.4.5 Participating Classes

MyManagedServerProc

EcPfManagedServer

GSO

EcAgManager

We also consider in the event trace the MSS SubAgent block to better clarify the system behavior.

4.5.1.5.4.6 Pre-conditions

The application is running and at least one server object is registered with the server.

4.5.1.5.4.7 Post-conditions

After the suspend command the application is in the suspended status. After the resume command, it is in the running status.

4.5.1.5.4.8 Scenario Description

The sub-agent manager, `EcAgManager`, calls the `PfSuspend` method to temporarily remove all objects from the server. This will unregister all objects, except the management object (`EcAg-Manager`), that are previously registered with the server.

At a later time, `PfResume` method is called to reregister all previously unregistered objects with the server.

4.5.1.5.4.9 Event Trace

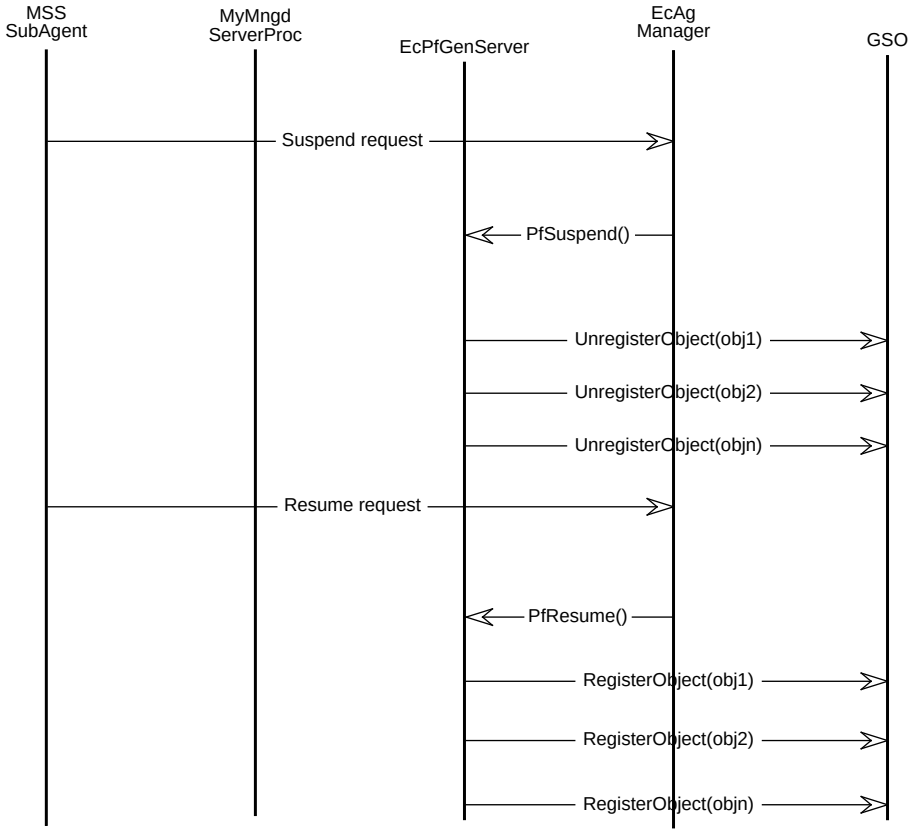

Figure 4.5.1.5-4 Process Framework Event Trace for Suspend and Resume

4.5.1.5.5 Scenario #5

4.5.1.5.5.1 Abstract

This scenario describes the logging of a generic event.

4.5.1.5.5.2 Interfaces

MSS.

4.5.1.5.5.3 Stimulus

An error/event is detected and is to be logged.

4.5.1.5.5.4 Desired Response

The error/event is stored in the appropriate log file.

4.5.1.5.5.5 Participating Classes

MyManagedServerProc

EcPfManagedServer

EcAgManager

EcAgEvent

We also consider in the event trace the MSS SubAgent block to better clarify the system behavior.

4.5.1.5.5.6 Pre-conditions

A log file is available.

4.5.1.5.5.7 Post-conditions

The event is stored in the log file.

4.5.1.5.5.8 Scenario Description

If an application object detects an event, it creates an instance of the `EcAgEvent` class which describes the event. It then passes the event object, along with the log type, to the `EcAgManager` to be logged using the method *`ProcessEvent`*.

4.5.1.5.5.9 Event Trace

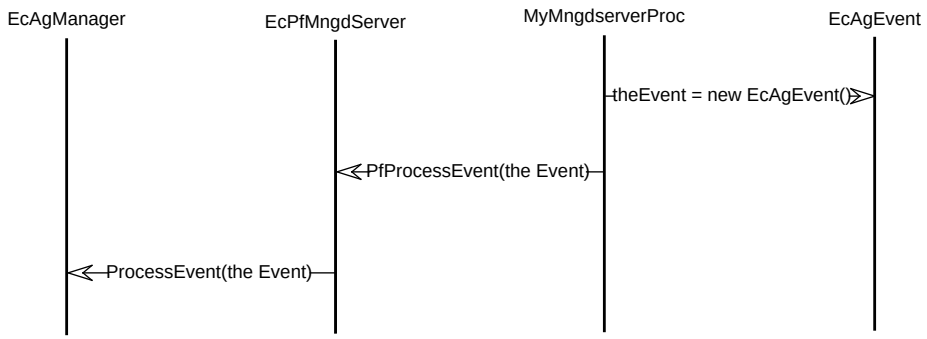

Figure 4.5.1.5-5. Process Framework Event Trace for Event/Error Logging

4.5.1.6 Implementation

The following observations can be made regarding the functionalities included in the framework:

- Some are specific to the distributed architecture chosen while others are independent of the architecture.
- Those specific to the distributed architecture are provided to the processes by appropriately using the classes provided by the distributed system.
- Those independent of the distributed architecture are being custom developed by the ECS subsystems.

Hence, the process framework is dependent on the underlying distributed architecture and the development of various infrastructural components by various ECS subsystems. The distributed architecture is currently being provided by using DCE/OODCE which is a COTS product. This is based on the recommendation that an encapsulated DCE (using OODCE) which provides security, object oriented capabilities, and a high degree of interoperability, be used in Release A & B time frame and migrate to a CORBA product in Release C. DCE is a very mature product which increasingly being used for client/server application development. OODCE provides the full object functionality that is needed to accomodate the object-oriented design employed by ECS.

4.5.1.7 Service/CSCI Management and Operation

4.5.1.7.1 System Management and Strategy

The whole Process Framework consists of a set of core services with distinct functionality to make the development of the distributed applications easier. Details are provided for each of these services in this document and other relevant ECS design documents. It comprises of only one CSC to accomodate the framework implementation. It is also comprised of the interface class CSCs from MSS.

4.5.1.7.2 Operator Interface

Not applicable.

4.5.1.7.3 Reports

None.

4.5.1.8 Frequently Asked Questions

4.5.2 Server Request Framework (SRF)

4.5.2.1 Overview

The SRF infrastructure is intended for use by the developers of ECS client applications to handle callbacks associated with asynchronous requests and notifications associated with subscriptions. The infrastructure standardizes these aspects of asynchronous communication.

The infrastructure is also intended for use by the developers of server applications. It will create and track service requests. It accepts callbacks (e.g., event notifications) and sends them to the corresponding client object. It automatically tracks objects which can handle incoming requests or accept callbacks and dispatches the requests or callbacks to the correct object, and matches up synchronous requests with their responses. Client applications can send requests to server objects identified by their UR - the infrastructure will resolve the UR to the correct server object. Finally, the infrastructure provides factories for creating server objects (and the client counter parts) dynamically.

4.5.2.2 Context

SRF will be used by ECS clients to access ECS services. It is intended to be supported by all servers that handle requests and thus by all clients that interact with those servers. It will not be used by MSS or other subsystems that do not need the request queuing and asynchronous features of SRF.

4.5.2.3 Object Model

SRF is structured into two layers. These two layers together provide the SRF functionality. The SRF Client/Server layer is used directly by applications. That will be shown first. The SRF Message Handling Layer is used internally by SRF to enhance the underlying CSS Messaging Layer. However, server developers will need to use some of the SRF Message Layer classes directly for initiating communication.

4.5.2.3.1 SRF Client Server Layer

See Figure 4.5.2.3.1-1.

4.5.2.3.1.1 AppAsynchRequest_C Class

Parent Class:EcSrAsynchRequest_C
Public:No
Distributed Object:No
Purpose and Description:
This object is not part of SRF, but rather an object that users of SRF are expected to create. This object should handle the client side of application specific SRF requests.

Attributes:

All Attributes inherited from parent class

Operations:

ClientOperation
Arguments:

Associations:

The AppAsynchRequest_C class has associations with the following classes:
None

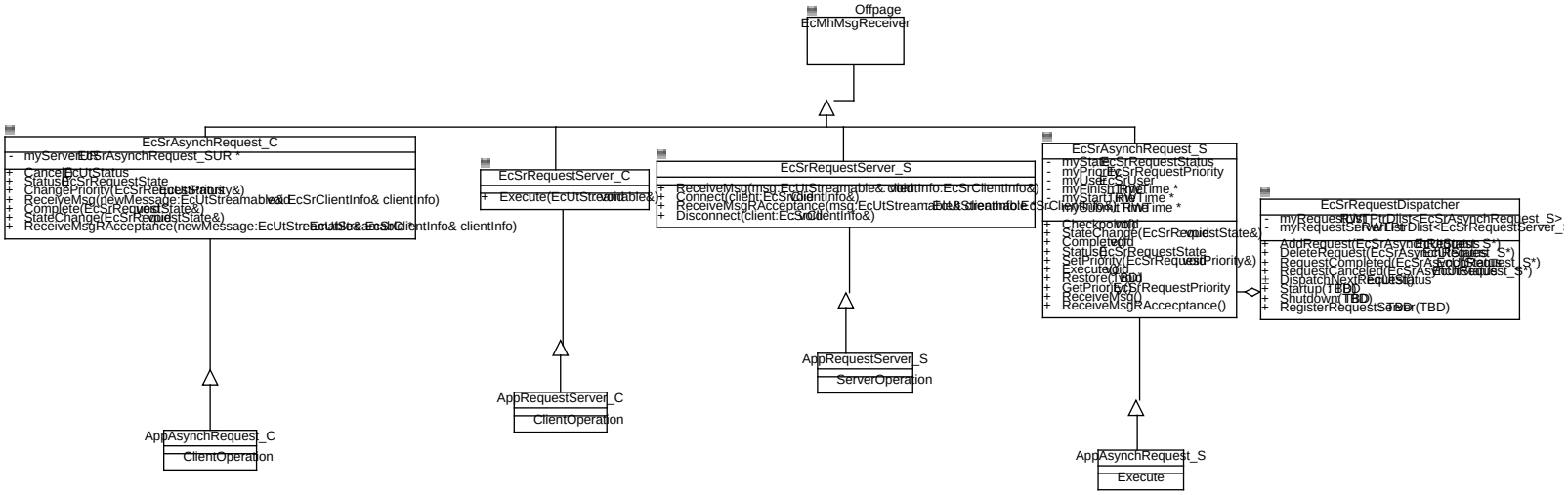

Figure 4.5.2.3.1-1. SRF Object Model

4.5.2.3.1.2 AppAsynchRequest_S Class

Parent Class:EcSrAsynchRequest_S
Public:No
Distributed Object:No
Purpose and Description:
This object is not part of SRF, but rather an object that users of SRF are expected to create. This object should handle the server side of application specific SRF requests.

Attributes:

All Attributes inherited from parent class

Operations:

Execute
Arguments:

Associations:

The AppAsynchRequest_S class has associations with the following classes:
None

4.5.2.3.1.3 AppRequestServer_C Class

Parent Class:EcSrRequestServer_C
Public:No
Distributed Object:No
Purpose and Description:
This object is not part of SRF, but rather an object that users of SRF are expected to create. This object should handle the client side of application specific SRF server connections.

Attributes:

All Attributes inherited from parent class

Operations:

ClientOperation
Arguments:

Associations:

The AppRequestServer_C class has associations with the following classes:
None

4.5.2.3.1.4 AppRequestServer_S Class

Parent Class:EcSrRequestServer_S
Public:No
Distributed Object:No
Purpose and Description:
This object is not part of SRF, but rather an object that users of SRF are expected to create. This object should handle the server side of application specific SRF server connections.

Attributes:

All Attributes inherited from parent class

Operations:

ServerOperation
Arguments:

Associations:

The AppRequestServer_S class has associations with the following classes:
None

4.5.2.3.1.5 EcMhMsgReceiver Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Provides general message receiving and response functions. Inherited by all classes which accept requests or call backs.

Attributes:

None

Operations:

None

Associations:

The EcMhMsgReceiver class has associations with the following classes:
None

4.5.2.3.1.6 EcSrAsynchRequest_C Class

Parent Class:EcMhMsgReceiver
Public:No
Distributed Object:No
Purpose and Description:
The corresponding asynchronous request object class for the client side. Created by client application prior to calling an asynchronous API.

Attributes:

myServerUR - identifies the request server object which is servicing the request.
Data Type:EcSrAsynchRequest_SUR *
Privilege:Private
Default Value:

Operations:

Cancel - Calling the method will result in the creation of an EcMhAsynchMessage object of the appropriate type, which is then submitted to the EcMhMessageHandler. The operation is synchronous, in that the client application's thread will be suspended until the response is received. This is handled transparently by the EcMhMessageHandler. Cancel() sends a Cancel request message to the corresponding EcMhASynchRequest_S object.
Arguments:
Return Type:EcUtStatus
Privilege:Public

ChangePriority - Similar to Cancel() accept that the request attempts to change the priority.
Arguments:EcSrRequestPriority&
Return Type:EcUtStatus
Privilege:Public

Complete - Called whenever the corresponding EcSrAsynchRequest_S object has completed the request.
Arguments:EcSrRequestState&
Return Type:void
Privilege:Public

ReceiveMsg - Specialization of method that handles dispatching for this class's methods.
Arguments:newMessage:EcUtStreamable& EcSrClientInfo& clientInfo
Return Type:void
Privilege:Public

ReceiveMsgRAcceptance - Specialization of method that handles dispatching for this class's methods.
Arguments:newMessage:EcUtStreamable& EcSrClientInfo& clientInfo
Return Type:EcUtStreamable *
Privilege:Public

StateChange - Called whenever the corresponding EcSrAsynchRequest_S object changes state.
Arguments:EcSrRequestState&
Return Type:void
Privilege:Public

Status - Similar to Cancel() accept that a Status message is sent to the server object.
Arguments:
Return Type:EcSrRequestState
Privilege:Public

Associations:

The EcSrAsynchRequest_C class has associations with the following classes:
None

4.5.2.3.1.7 EcSrAsynchRequest_S Class

Parent Class:EcMhMsgReceiver
Public:No
Distributed Object:No
Purpose and Description:
Standard mechanism class for controlling asynch remote operations. Provides request control operations (e.g. Cancel) as well as asynch notification operations (e.g. Complete, StateChange). Created when a new request is received by the server.

Attributes:

myFinishTime - The time when the request processing was completed.
Data Type:RWTime *
Privilege:Private
Default Value:

myPriority - The request priority. Note: the default prioritization scheme is fairly simpleminded. The RequestDispatcher class can be specialized to add more complex priorities.
Data Type:EcSrRequestPriority
Privilege:Private
Default Value:

myStartTime - The time when processing of the request began.
Data Type:RWTime *
Privilege:Private
Default Value:

myState - The current state of the request.
Data Type:EcSrRequestStatus
Privilege:Private
Default Value:

mySubmitTime - The time that the request was submited.
Data Type:RWTime *
Privilege:Private
Default Value:

myUser - The user that submitted the request.
Data Type:EcSrUser
Privilege:Private
Default Value:

Operations:

Checkpoint
Arguments:
Return Type:void
Privilege:Public

Complete
Arguments:
Return Type:void
Privilege:Public

Execute
Arguments:
Return Type:void
Privilege:Public

GetPriority
Arguments:
Return Type:EcSrRequestPriority
Privilege:Public

ReceiveMsg
Arguments:
Return Type:Void
Privilege:Public

ReceiveMsgRAccecptance
Arguments:
Return Type:Void
Privilege:Public

Restore
Arguments:TBD
Return Type:void
Privilege:Public

SetPriority
Arguments:EcSrRequestPriority&
Return Type:void
Privilege:Public

StateChange
Arguments:EcSrRequestState&
Return Type:void
Privilege:Public

Status
Arguments:
Return Type:EcSrRequestState
Privilege:Public

Associations:

The EcSrAsynchRequest_S class has associations with the following classes:
EcSrRequestDispatcher (Aggregation)

4.5.2.3.1.8 EcSrRequestDispatcher Class

Parent Class:Not Applicable
Public:No
Distributed Object:No

Purpose and Description:

Attributes:

myRequestList - The list of requests currently known to the server
Data Type:RWTPtrDlist<EcSrAsynchRequest_S>
Privilege:Private
Default Value:

myRequestServerList - The list of all server objects currently known to the system.
Data Type:RWTPtrDlist<EcSrRequestServer_S>
Privilege:Private
Default Value:

Operations:

AddRequest - Adds a Request to the list of open Requests.
Arguments:EcSrAsynchRequest_S*
Return Type:EcUtStatus
Privilege:Public

DeleteRequest - Removes a Request from the list of open Requests.
Arguments:EcSrAsynchRequest_S*
Return Type:EcUtStatus
Privilege:Public

DispatchNextRequest - Dispatches next available request based on priority.
Arguments:
Return Type:EcUtStatus
Privilege:Protected

RegisterRequestServer - Register a new RequestServer with the dispatcher.
Arguments:TBD
Return Type:TBD
Privilege:Public

RequestCanceled - Indicates that the request has been cancelled. My result in client notification.
Arguments:EcSrAsynchRequest_S*
Return Type:EcUtStatus
Privilege:Public

RequestCompleted - Indicates that the request has been finished. My result in client notification.

Arguments:EcSrAsynchRequest_S*
Return Type:EcUtStatus
Privilege:Public

Shutdown - Shutdown routine. Details TBD.
Arguments:TBD
Return Type:TBD
Privilege:Public

Startup - Initialization routine. Details TBD.
Arguments:TBD
Return Type:TBD
Privilege:Public

Associations:

The EcSrRequestDispatcher class has associations with the following classes:
None

4.5.2.3.1.9 EcSrRequestServer_C Class

Parent Class:EcMhMsgReceiver
Public:No
Distributed Object:No
Purpose and Description:
Generic base class for the client portion of all distributed classes. Provides client wrapper around asynch messaging framework. Also responsible for maintaining the connection to the corresponding server object. In case of a Server Process failure, will re-establish connection with server object.

Attributes:

All Attributes inherited from parent class

Operations:

Execute - Sends an asynchronous request message to the corresponding EcSrRequestServer_S object for execution. Derived types of EcSrRequestServer_C will provide server specific methods (e.g. Search). For each method the derived type will construct an asychronous request message and transmit it to the server via this method.
Arguments:EcUtStreamable&
Return Type:void

Privilege:Public

Associations:

The EcSrRequestServer_C class has associations with the following classes:
None

4.5.2.3.1.10 EcSrRequestServer_S Class

Parent Class:EcMhMsgReceiver
Public:No
Distributed Object:No
Purpose and Description:
Generic base class for the server portion of all distributed classes. Provides server wrapper around messaging framework. Also responsible for maintaining the connection to the corresponding client object (EcSrRequestServer_C).

Attributes:

All Attributes inherited from parent class

Operations:

Connect - Receives initial connect() request from client application.
Arguments:client:EcSrClientInfo&
Return Type:void
Privilege:Public

Disconnect - Handles disconnect() request from client application.
Arguments:client:EcSrClientInfo&
Return Type:void
Privilege:Public

ReceiveMsg - Overrides the virtual method in the EcMhMessageReceiver base class. Whenever EcMhMessageHandler receives a message for this object, it will call this ReceiveMsg method. The validity of the type of request message for the particular request server has already been validated before the MessageHandler calls the request server.
Arguments:msg:EcUtStreamable& clientInfo:EcSrClientInfo&
Return Type:void
Privilege:Public

ReceiveMsgRAcceptance - Similar to ReceiveMsg accept handles the case of waiting for acceptance replies.
Arguments:msg:EcUtStreamable& clientInfo:EcSrClientInfo&

Return Type:EcUtStreamable *
Privilege:Public

Associations:

The EcSrRequestServer_S class has associations with the following classes:
None

4.5.2.3.2 SRF Message Handling Layer

See Figure 4.5.2.3.2-1.

4.5.2.3.2.1 EcMhMsgEnvelope Class

Parent Class:EcUtStreamable
Public:No
Distributed Object:No
Purpose and Description:
Simple wrapper around the application message to track originator/destination of a message. Holds extra information about the message: sender, receiver, etc.

Attributes:

myAcceptanceExpected - Flag indicating if an acceptance message is expected.
Data Type:EcTBoolean
Privilege:Private
Default Value:

myLetter - Stores the application message to be sent.
Data Type:EcUtStreamable
Privilege:Private
Default Value:

myLettersID - The UR of the letter.
Data Type:EcUtClassID
Privilege:Private
Default Value:

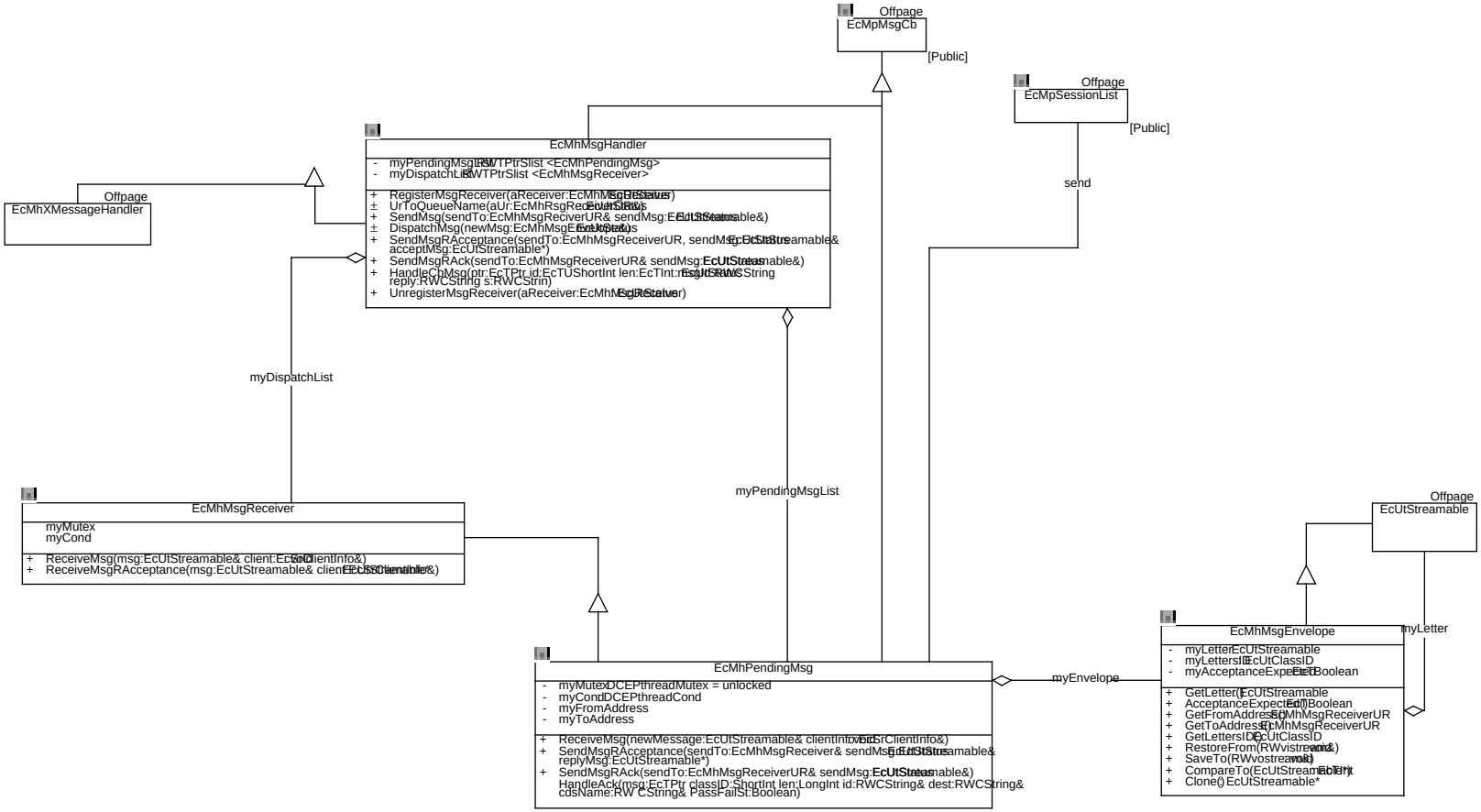

Figure 4.5.2.3.2-1: SRF Message Object Model

Operations:

AcceptanceExpected - returns myAcceptanceExpected. True if an acceptance response is expected.
Arguments:
Return Type:EcTBoolean
Privilege:Public

Clone
Arguments:
Return Type:EcUtStreamable*
Privilege:Public

CompareTo
Arguments:EcUtStreamable*
Return Type:EcTInt
Privilege:Public

GetFromAddress - returns myFromAddress.
Arguments:
Return Type:EcMhMsgReceiverUR
Privilege:Public

GetLetter - returns myLetter() reference.
Arguments:
Return Type:EcUtStreamable
Privilege:Public

GetLettersID - returns myLetterID.
Arguments:
Return Type:EcUtClassID
Privilege:Public

GetToAddress - returns myToAddress.
Arguments:
Return Type:EcMhMsgReceiverUR
Privilege:Public

RestoreFrom
Arguments:RWvistream&
Return Type:void
Privilege:Public

SaveTo
Arguments:RWvostream&
Return Type:void
Privilege:Public

Associations:

The EcMhMsgEnvelope class has associations with the following classes:
EcMhPendingMsg (Aggregation)

4.5.2.3.2.2 EcMhMsgHandler Class

Parent Class:EcMpMsgCb
Public:No
Distributed Object:No
Purpose and Description:
Central manager for all message communications. Provides the following: ·Wrapper around Asynch Messaging Framework. ·Conversion of UR's to Named Queue ·Maintains list of EcSrRequestServer_S objects and their UR's ·Dispatches in-coming messages to appropriate EcSrRequestServer_S. ·Routes out-going messages to proper named queues in asynch messaging framework. Created at application start-up time by process framework

Attributes:

myDispatchList - List of registered EcMhMsgReceiver objects. The list is used to dispatch incoming messages.
Data Type:RWTPtrSlist <EcMhMsgReceiver>
Privilege:Private
Default Value:

myPendingMsgList - List of pending messages via EcMhPendingMsg objects. The pending messages are those that are awaiting Replies or Acks.
Data Type:RWTPtrSlist <EcMhPendingMsg>
Privilege:Private
Default Value:

Operations:

DispatchMsg
Arguments:newMsg:EcMhMsgEnvelope&
Return Type:EcUtStatus
Privilege:Protected

HandleCbMsg
Arguments:ptr:EcTPtr id:EcTUShortInt len:EcTInt msgId:RWCString reply:RWCString s:RWCStrin
Return Type:EcUtStatus
Privilege:Public

RegisterMsgReceiver
Arguments:aReceiver:EcMhMsgReceiver
Return Type:EcUtStatus
Privilege:Public

SendMsg
Arguments:sendTo:EcMhMsgReciverUR& sendMsg:EcUtStreamable&
Return Type:EcUtStatus
Privilege:Public

SendMsgRAcceptance
Arguments:sendTo:EcMhMsgReceiverUR, sendMsg:EcUtStreamable& acceptMsg:EcUtStreamable*
Return Type:EcUtStatus
Privilege:Public

SendMsgRAck
Arguments:sendTo:EcMhMsgReceiverUR& sendMsg:EcUtStreamable&
Return Type:EcUtStatus
Privilege:Public

UnregisterMsgReceiver
Arguments:aReceiver:EcMhMsgReceiver
Return Type:EcUtStatus
Privilege:Public

UrToQueueName
Arguments:aUr:EcMhRsgReceiverUR&
Return Type:EcUtStatus
Privilege:Protected

Associations:

The EcMhMsgHandler class has associations with the following classes:
None

4.5.2.3.2.3 EcMhMsgReceiver Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Provides general message receiving and response functions. Inherited by all classes which accept requests or call backs.

Attributes:

myCond

myMutex

Operations:

ReceiveMsg
Arguments:msg:EcUtStreamable& client:EcSrClientInfo&
Return Type:void
Privilege:Public

ReceiveMsgRAcceptance
Arguments:msg:EcUtStreamable& client:EcSrClientInfo&
Return Type:EcUtStramable*
Privilege:Public

Associations:

The EcMhMsgReceiver class has associations with the following classes:
EcMhMsgHandler (Aggregation)

4.5.2.3.2.4 EcMhPendingMsg Class

Parent Class:EcMhMsgReceiver
Parent Class:EcMpMsgCb
Public:No
Distributed Object:No
Purpose and Description:
Used by EcMhMsgHandler to implement SendMsgRAcceptance and SendMsgRack. This class takes care of the wait for reply or ack portion of the Send. It uses myCond & myMutex for the waits.

Attributes:

myCond - Condition variable used for signaling with myMutex. Together, they are used for signaling the receipt of a response message.
Data Type:DCEPthreadCond
Privilege:Private
Default Value:

myFromAddress
Data Type:
Privilege:Private
Default Value:

myMutex - Mutex used for locking with myCond. Together, they are used for signaling the receipt of a response message.
Data Type:DCEPthreadMutex
Privilege:Private
Default Value:unlocked

myToAddress
Data Type:
Privilege:Private
Default Value:

Operations:

HandleAck
Arguments:msg:EcTPtr classID:ShortInt len:LongInt id:RWCString& dest:RWCString& cdsName:RW CString& PassFailSt:Boolean

ReceiveMsg
Arguments:newMessage:EcUtStreamable& clientInfo:EcSrClientInfo&
Return Type:void
Privilege:Public

SendMsgRAcceptance
Arguments:sendTo:EcMhMsgReceiver& sendMsg:EcUtStreamable& replyMsg:EcUtStreamable*
Return Type:EcUtStatus
Privilege:Public

SendMsgRAck
Arguments:sendTo:EcMhMsgReceiverUR& sendMsg:EcUtStreamable&
Return Type:EcUtStatus

Privilege:Public

Associations:

The EcMhPendingMsg class has associations with the following classes:
Class: EcMpSessionList send
EcMhMsgHandler (Aggregation)

4.5.2.3.2.5 EcMhXMessageHandler Class

Parent Class:EcMhMsgHandler
Public:No
Distributed Object:No
Purpose and Description:
Specialized version of EcSrMessageHandler for use in X Windows applications. Ensures that all message dispatching is performed within the X thread. Created at application start-up time (mechanism TBD)

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The EcMhXMessageHandler class has associations with the following classes:
None

4.5.2.3.2.6 EcMpMsgCb Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class will handle two types of callbacks: 1. For ordinary receive messages: If an ordinary message is received, then HandleMsg is invoked; 2. For acknowledgment of messages: If an acknowledgement 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. A list of sending sessions is maintained in a given application. A sender session list contains a callback object which provides virtual funtions to be called when a send is complete. This is done at the sender

side. 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)

Attributes:

None

Operations:

None

Associations:

The EcMpMsgCb class has associations with the following classes:
None

4.5.2.3.2.7 EcMpSessionList Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This is a container class whose element type is a logical name and will inherit from the RWTPtrSlist class. 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.

Attributes:

None

Operations:

None

Associations:

The EcMpSessionList class has associations with the following classes:
Class: EcMhPendingMsg send

4.5.2.3.2.8 EcUtStreamable Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The EcUtStreamable class has associations with the following classes:
EcMhMsgEnvelope (Aggregation)

4.5.2.4 Functional Model

The following figure shows the software layers in a typical SRF based application. The SRF Framework provides two layers of the application: The client/server layer and the Message Handling layer. These layers build on the message passing (implementation one) layer provided by the CSS subsystem.

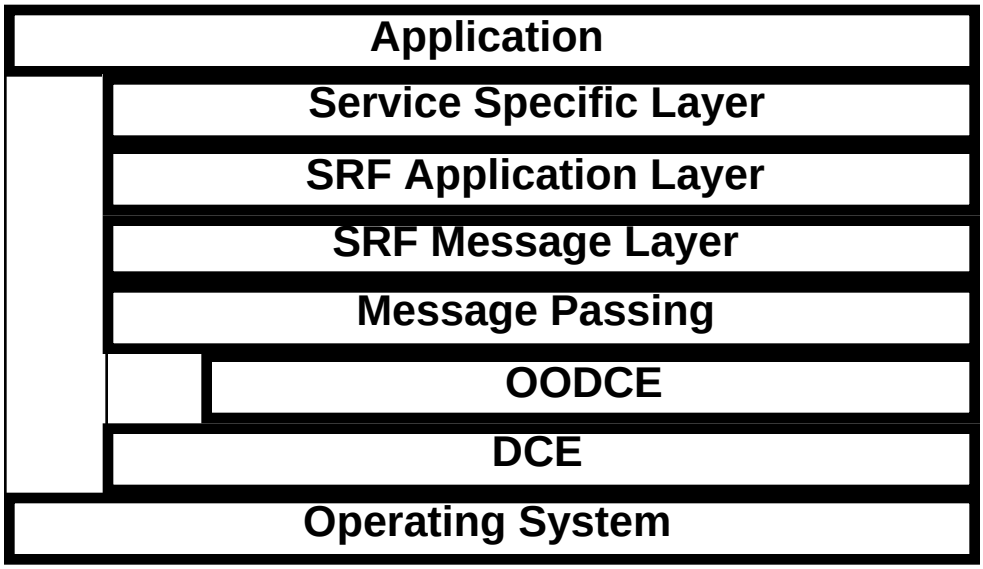

Figure 4.5.2.4-1. SRF Layering

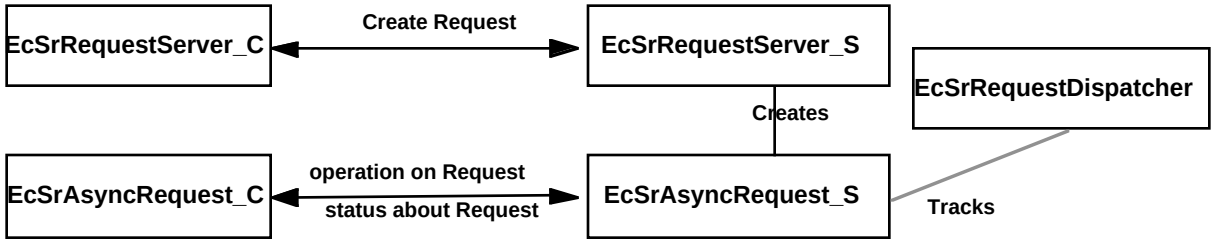

Figure 4.5.2.4-2. SRF Conceptual View

The SRF Client/Server layer provides a set of 5 generic object classes for both clients and servers. The figure below presents a high-level overview of the framework.

The infrastructure provides

- generic asynchronous request object classes for the client and server side
- object classes which implement the communication between client and server and dispatch requests or callbacks to the correct client or server object
- a generic message class for transmitting requests, responses, and callbacks

There will be two client side implementations of the infrastructure: one for clients which operate in an X-windows environment; and one for other types of clients. The special version for X-Windows is needed because X-windows and Motif do not support multi-threading. The infrastructure will insulate the X environment from the threads which are necessary to support asynchronous communication.

4.5.2.4.1 Fundamentals

The infrastructure includes object classes (named EcSr.....) which provide a level of abstraction above the Messaging framework of CSS.

Objects on the server side which accept requests from clients are specialized from a generic RequestServer class. The RequestServer class knows how to accept and track requests and how to interface with the message infrastructure. There is a corresponding RequestServer_C object class for the client side. Server developers specialize it to provide the client API for each RequestServer class. The client application calls a RequestServer_C object to send requests to the corresponding server object. The generic RequestServer_C class knows how to package a call into a request and interface with the message infrastructure.

RequestServer_S and RequestServer_C are local object classes, i.e., they are not distributed objects.

All requests to servers pass through the Messaging framework. The infrastructure includes MessageHandler objects for clients and servers which provide the direct interface to the Messaging framework.

The MessageHandler offers interfaces for sending a request requiring a reply; sending a request requiring an acknowledgment; and sending a notification requiring no confirmation at all. The MessageHandler tracks requests for which replies or acknowledgments are pending.

Each asynchronous requests are tracked by an AsynchRequest object. The client application must supply an AsynchRequest object when it makes an asynchronous call to a RequestServer_C. The AsynchRequest object provides standard functions to accept callbacks from the server indicating request completion or a change in the request status.

Each instance of an AsynchRequest will have a unique identification for reference by the client and server For the purpose of distributed reference, the unique identifier is turned into a UR.

All objects which expect callbacks or requests, register themselves with the MessageHandler. AsynchRequest objects register themselves automatically when the request is submitted to the server. Each request, response, and callback passing between a client and a server requires an AsynchMessage object of a unique type.

4.5.2.5 Dynamic Model Scenarios

4.5.2.5.1 Scenerio #1: Client View

4.5.2.5.1.1 Abstract

This scenario demonstrates the initialization sequence of a single SRF AsynchRequest. In this hypothetical example, a server writer has created a new service called "sleeper". The sleeper service supports a single operation: sleep. This scenario covers the client's view. The server's view follows.

4.5.2.5.1.2 Interfaces

SleepRequestServer_C & SleepAsynchRequest_C.

4.5.2.5.1.3 Stimulus

Clients call the sleep method on the client side object (SleepRequestServer_C).

4.5.2.5.1.4 Desired Response

The server receives the sleep request message & queues it into its dispatch queue. At some later time the client will be notified that the sleep call has completed.

4.5.2.5.1.5 Participating Classes

SleepAsynchRequest_(C & S), SleepRequestServer_(C & S), SleepRequestMsg, EcMhMsgHandler, EcMhPendingMsg, and EcMhMsgEnvelope.

4.5.2.5.1.6 Pre-conditions

The client has already constructed the SleepRequestServer_C.

4.5.2.5.1.7 Post-conditions

After the client is notified, the server returns to its original state & the request is considered complete.

4.5.2.5.1.8 Scenario Description

1. A client application calls SleepRequestServer_C::Sleep() (not shown).
2. The sleep() method then constructs a message and sends it via EcMhMsgHandler::SendMsgRAcceptance().
3. The SRF communication layer then constructs a pending message object (to hold the outstanding message). That object in turn, constructs an envelope.
4. The Msghandler then calls SendMsgRAcceptance() on the pending message object. which in turn uses the underlying CSS Messaging call: EcMpQueueOut::Send(). At this point, the message is on its way to the server.
5. After a brief delay, the response comes from the server. This causes the MsgHandler to construct the SleepAsynchRequest_C object. Control is then returned to SleepRequestServer_C which then returns control to the client application.

4.5.2.5.1.9 Event Trace

See Figure 4.5.2.5-1.

4.5.2.5.2 Scenario #2: Server View

4.5.2.5.2.1 Abstract

This scenario demonstrates the initialization sequence of a single SRF AsynchRequest. In this hypothetical example, a server writer has created a new service called "sleeper". The sleeper service supports a single operation: sleep. This scenario is the server's view.

4.5.2.5.2.2 Interfaces

SleepRequestServer_S & SleepAsynchRequest_S.

4.5.2.5.2.3 Stimulus

Server receives sleep request.

4.5.2.5.2.4 Desired Response

The server receives the sleep request message & queues it into its dispatch queue. At some later time the client will be notified that the sleep call has completed.

4.5.2.5.2.5 Participating Classes

SleepAsynchRequest_(C & S), SleepRequestServer_(C & S), SleepRequestMsg, EcMhMsgHandler, EcMhPendingMsg, EcSrReuqestDispatcher, and EcMhMsgEnvelope.

4.5.2.5.2.6 Pre-conditions

The server has already constructed the SleepRequestServer_S and initialized SRF.

4.5.2.5.2.7 Post-conditions

After the client is notified, the server returns to its original state & the request is considered complete.

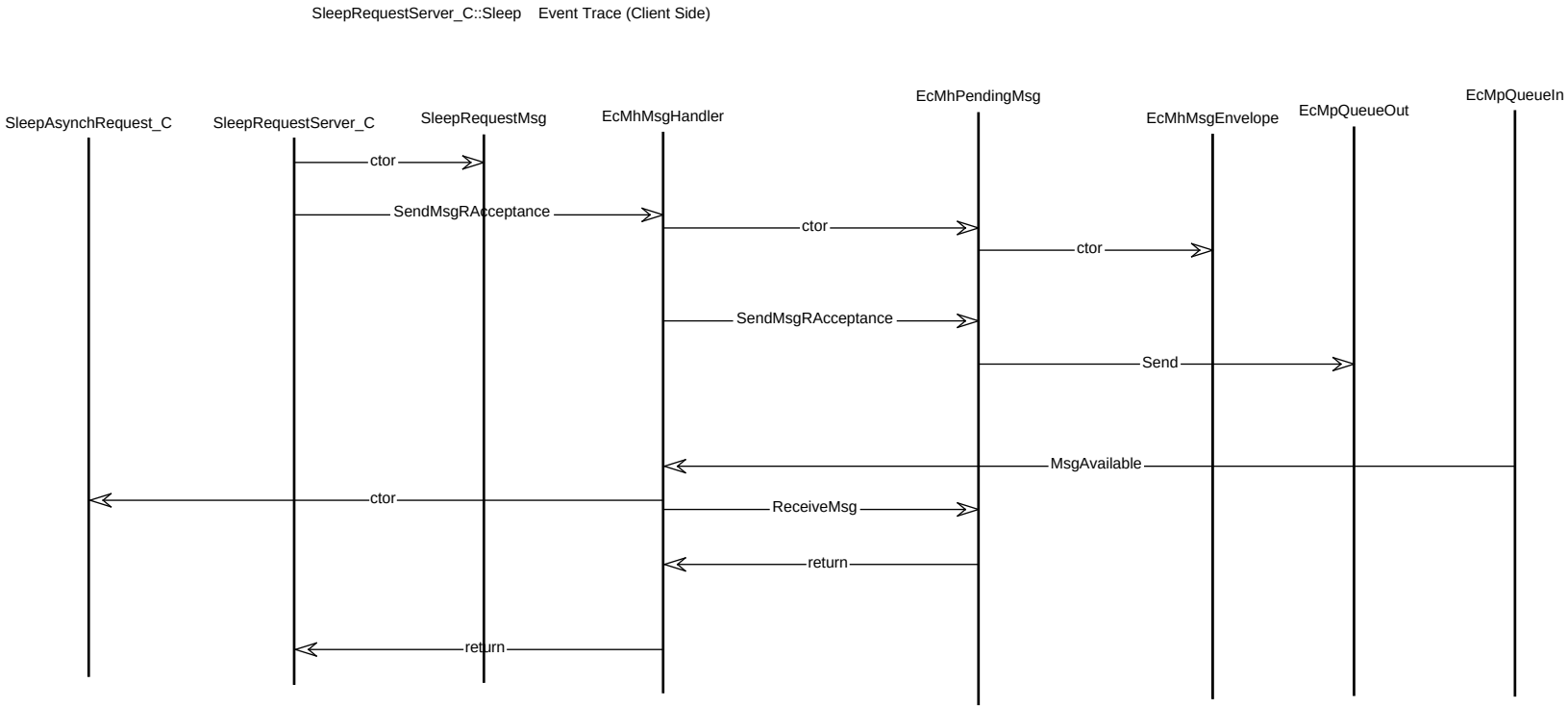

Figure 4.5.2.5-1. SRF Event Trace

4.5.2.5.2.8 Scenario Description

1. The message is received by the EcMpQueueIn object (from CSS Message Passing).
2. EcMpQueueIn passes the message to EcMhMsgHandler which then looks up in its internal dispatch table the object that should be receiving a message with this UR & type.
3. The SleepRequestServer_S::ReceiveMsgRAcceptance is invoked. That method was overridden by the server writer. The message is then dispatched to the correct method (in this case sleep()).
4. The SleeperRequestServer_S object constructs a SleepAsynchRequest_S object to manage the asynch request and then submits it to the EcSrRequestDispatcher which queues the message for later execution.
5. The SleepRequestServer_S then returns a message which will be sent to the client. That message contains the UR for the SleepAsyncRequest_C object.
6. When the dispatcher executes the sleep request at a later time, the client will be notified. See the following scenario.

4.5.2.5.2.9 Event Trace

See Figure 4.5.2.5-2.

4.5.2.5.3 Scenerio #3: Client Notification

4.5.2.5.3.1 Abstract

This scenario demonstrates the notification process of SRF. In this hypothetical example, a server writer has created a new service called "sleeper". The sleeper service supports a single operation: sleep. The client now receives the notification that the sleep has been completed. The server's view is covered in the next scenario.

4.5.2.5.3.2 Interfaces

SleepAsynchRequest_C.

4.5.2.5.3.3 Stimulus

The server sends to the client the sleep notification.

4.5.2.5.3.4 Desired Response

The client dequeues the message and it is dispatched to the SleepAsynchRequest_C object.

4.5.2.5.3.5 Participating Classes

SleepAsynchRequest_(C & S), EcMhMsgHandler, EcMhMsgReceiver.

4.5.2.5.3.6 Pre-conditions

The client has previously called sleep() on the SleepAsychRequest_C object.

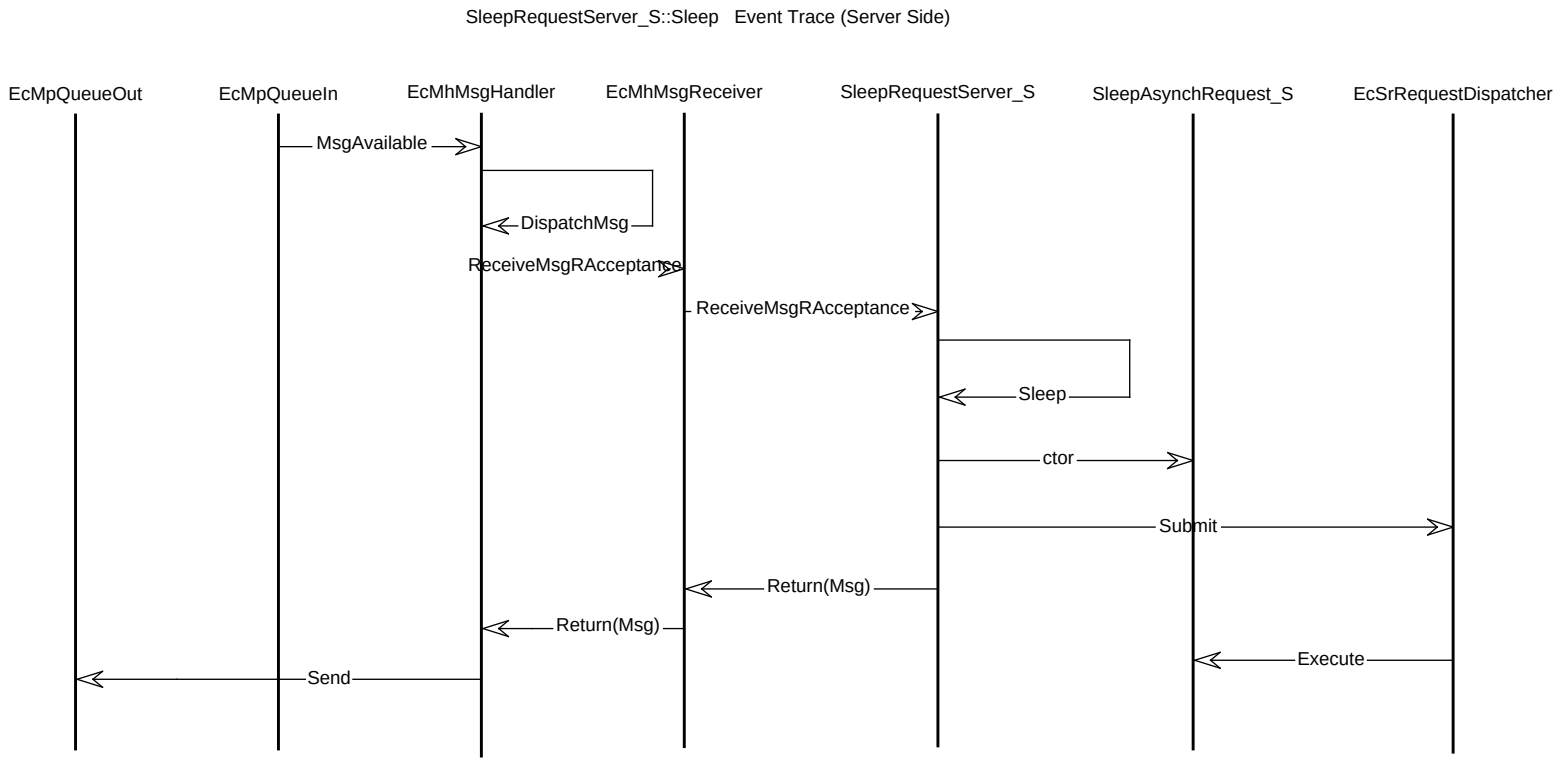

Figure 4.5.2.5-2. SRF Event Trace

4.5.2.5.3.7 Post-conditions

After the client is notified, the server returns to its original state & the request is considered complete.

4.5.2.5.3.8 Scenario Description

1. The notification message is passed to EcMhMsgHandler.
2. EcMhMsgHandler determines the object that should receive this message based on its UR. The ReceiveMsg() method is then invoked on the SleepAsynchRequest_C object.
3. The SleepAsynchRequest_C object then dispatches the message to the correct method. In this case that is TimeIntervalNofification(). That method might then perform some service specific action.

4.5.2.5.3.9 Event Trace

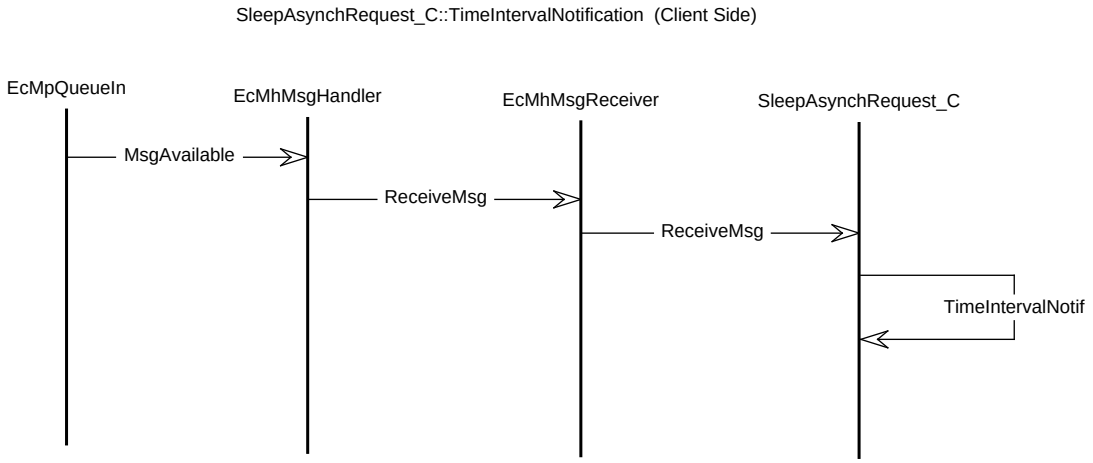

Figure 4.5.2.5-3. SRF Event Trace

4.5.2.5.4 Scenerio #4: Client Notification - Server View

4.5.2.5.1.1 Abstract

This scenario demonstrates the notification process of SRF. In this hypothetical example, a server writer has created a new service called "sleeper". The sleeper service supports a single operation: sleep. The client now receives the notification that the sleep has been completed. This is the server's view.

4.5.2.5.1.2 Interfaces

SleepAsynchRequest_S.

4.5.2.5.1.3 Stimulus

A timer expires based on a request which is currently being processed. In this example, the "processing" is merely waiting for the timer.

4.5.2.5.1.4 Desired Response

The server notifies the client that the sleep call has completed.

4.5.2.5.1.5 Participating Classes

SleepAsynchRequest_(C & S), EcMhMsgHandler, EcMhPendingMsg, and EcMhMsgEnvelope.

4.5.2.5.1.6 Pre-conditions

The server has already dispatched the sleep request message.

4.5.2.5.1.7 Post-conditions

After the client is notified, the server returns to its original state & the request is considered complete.

4.5.2.5.1.8 Scenario Description

1. The sleep request is being processed. When the timer expires, it calls SleepAsynchRequest_S's TimeIntervalNotification().
2. The server method (TimeIntervalNotification()) then constructs a message of the appropriate type and sends it using the EcMhMsgHandler::SendMsgRAck().
3. Since the message requires an ack, the message handler constructs a EcMhPendingMsg object and in turn a EcMhMsgEnvelope object to hold the message.
4. The SendMsgWAck() method is called which in turn, invokes the Send() method on the EcMpQueueOut object to actually send the message across the network. The server thread then waits for the ack.
5. When the ack arrives, the handleAck() method is invoked on the EcMhPendingMsg object. The EcMhPendingMsg object then returns control to the handler since the ack has been received.
6. The handler object then returns the status value to the SleepAsynchRequest_S object indicating success. That object might then reset the timer for a future time and notification.

4.5.2.5.1.9 Event Trace

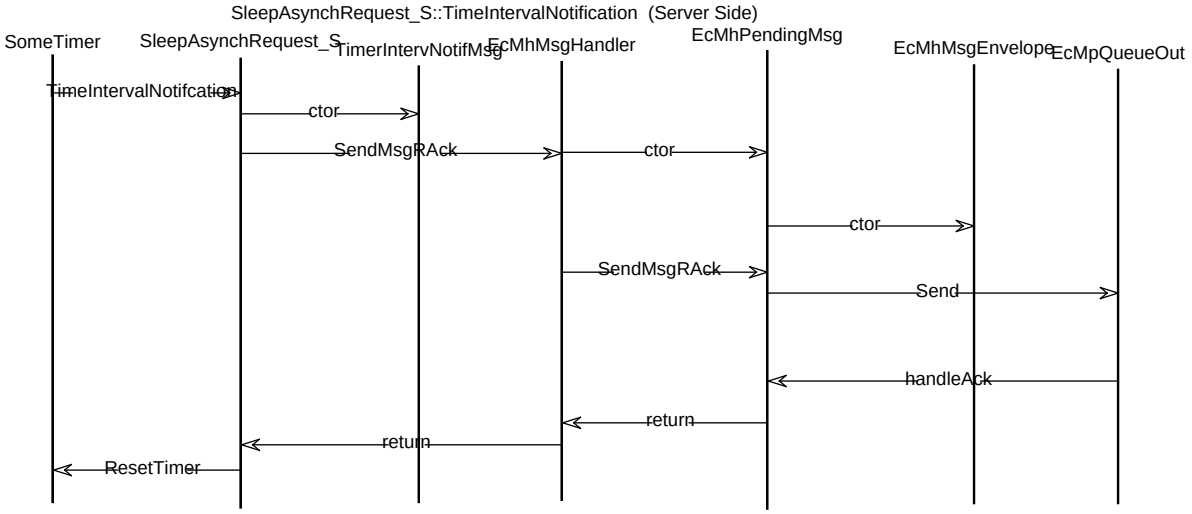

Figure 4.5.2.5-4. SRF Event Trace

4.5.3 Subscription Server

4.5.3.1 Overview

The role of the Subscription Server is to support the detection of previously defined events and to perform specified actions on behalf of clients who have previously registered to those events. In order to actually perform its role in the ECS, the Subscription Server Process requires that its clients be active and have previously registered events with it. Examples of events include science granule insertion, metadata update, new advertisement, new schema export to DDICT, etc.

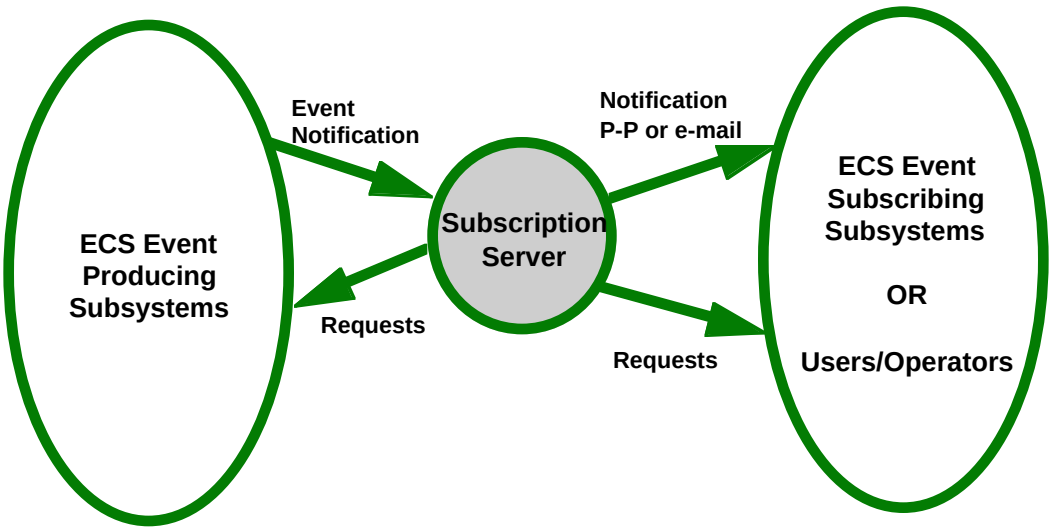

Figure 4.5.3.1-1. Subscription Server Context

4.5.3.2 Context

The subscription server addresses the need for a common mechanism across subsystems which can support a generic event-action model based on a event producer/consumer paradigm. Some of the users of the subscription server in Release B are listed below:

ECS Subscription Event Producers:

- Data Server
- Advertising Service and
- Data

ECS Subscription Users or Submitters:

- Client on behalf of science users or operators
- Data Processing
- Planning

4.5.3.3 Object Model, Part 1

See Figure 4.5.3.3-1.

4.5.3.3.1 EcClAction Class

Parent Class:EcShActionBase
Public:Yes
Distributed Object:No
Purpose and Description:
A client interface object that represents the components of the action to be performed when a subscription is triggered. The possibilities are that the client will receive a notification (including all parameters that are returned by the object that triggers the subscription and an optional piece of client-specified text) and/or a request that will be executed. The client is required to specify an action for each subscription.

Attributes:

myRequest - The request that is currently associated with this action.
Data Type:EcClRequest
Privilege:Private
Default Value:

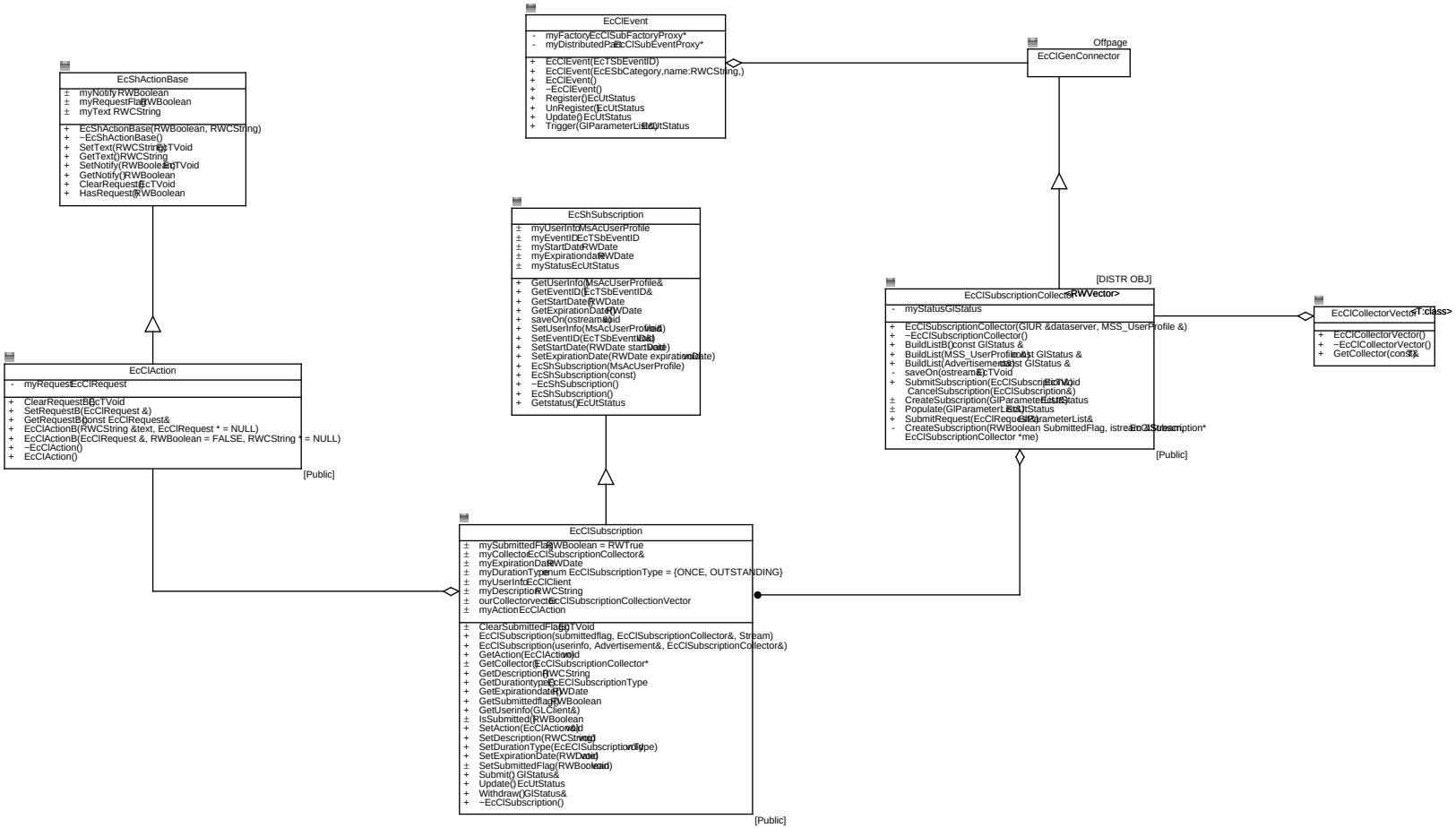

Figure 4.5.3.3-1. Subscription Server Object Model, Part 1

Operations:

ClearRequestB - Used to clear any request that has been set for the action.
Arguments:
Return Type:EcTVoid
Privilege:Public

EcCIAction - Default constructor
Arguments:
Return Type:Void
Privilege:Public

EcClActionB - Used to construct an action from a piece of text and, RELEASE B: optionally, a request. The notification flag is set.
Arguments:RWCString &text, EcClRequest * = NULL
Return Type:Void
Privilege:Public

EcClActionB - Used to construct an action from a request and, optionally, a value for the notification flag and a piece of text.
Arguments:EcClRequest &, RWBoolean = FALSE, RWCString * = NULL
Return Type:Void
Privilege:Public

GetRequestB - Returns the request currently set for the action.
Arguments:
Return Type:const EcClRequest&
Privilege:Public

SetRequestB - Sets the request to be executed when the subscription fires.
Arguments:EcClRequest &
Return Type:Void
Privilege:Public

~EcClAction - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClAction class has associations with the following classes:
EcClSubscription (Aggregation)

4.5.3.3.2 EcClCollectorVector Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
This private class is maintained in static memory by EcCl items (EcClSubscription and EcClESDTReference). Those items contain a EcClCollectorVector. This class supports the "hidden construction" of collector objects (and hence, connections to a dataserver). It is a RWTPtrOrderedVector, which is parameterized so that the item which is maintaining it can decide what type of collector to store and track. This class represents a table of available EcCl<item>Collectors which were created by item-level EcCl public classes (i.e., EcClSubscription or EcClESDTReference). This table will contain a pointer to one EcCl<item>Collector per dataserver. All item-level objects created in the client space independently of a client-created EcCl<item>Collector will be assigned to the default EcCl<item>Collector as defined by the entry in the EcClCollectorVector which corresponds to the dataserver for which the client is creating the independent item. (phew!)

Attributes:

None

Operations:

EcClCollectorVector - This is the default constructor, for initial startup.
Arguments:
Return Type:Void
Privilege:Public

GetCollector - This operation finds the existing collector for the given dataserver (if any) or creates one if none exists. It returns the reference to the chosen collector. Because the class is a template, the return value will match (be the right kind of collector for) the object which is calling this operation.
Arguments:const
Return Type:T&
Privilege:Public

~EcClCollectorVector - The destructor should presumably be called only when the vector is empty, but it might be a good idea to check for any collectors here and close them down.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClCollectorVector class has associations with the following classes:
None

4.5.3.3.3 EcClEvent Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myDistributedPart - The distributed part of CsClEvent
Data Type:EcClSubEventProxy*
Privilege:Private
Default Value:

myFactory - This is a pointer to the subscription server's factory distributed object. This provides client interface to the subscription server's factory distributed object, and is used to create client connection and request objects.
Data Type:EcClSubFactoryProxy*
Privilege:Private
Default Value:

Operations:

EcClEvent - full constructor, creates auto-filled-in new event object
Arguments:EcESbCategory,name:RWCString,
Return Type:Void
Privilege:Public

EcClEvent - Usual constructor, takes event ID and instantiates the event
Arguments:EcTSbEventID
Return Type:Void
Privilege:Public

EcClEvent - Defualt constructor
Arguments:
Return Type:Void
Privilege:Public

Register - Put this event into the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

Trigger - Activate this event in the system such that all subscriptions to this event will be processed
Arguments:GlParameterList&
Return Type:EcUtStatus
Privilege:Public

UnRegister - Take this event out of the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

Update - Change this event's value in the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

~EcClEvent - Destructor, cleans up and removes this object
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClEvent class has associations with the following classes:
None

4.5.3.3.4 EcClGenConnector Class

Parent Class:Not Applicable

Attributes:

None

Operations:

None

Associations:

The EcClGenConnector class has associations with the following classes:
EcClEvent (Aggregation)

4.5.3.3.5 EcClSubscription Class

Parent Class:EcShSubscription
Public:Yes
Distributed Object:No
Purpose and Description:
This class is the client side subscription which can either be created from advertisements or from exisiting subscriptions from the server side (through a stream.)

Attributes:

myAction - The action to be performed when the subscription fires.
Data Type:EcClAction
Privilege:Protected
Default Value:

myCollector - A pointer to the collector that this reference is a member of. If this pointer is null, then this reference is a member of one of the collectors in the static collector vector.
Data Type:EcClSubscriptionCollector&
Privilege:Protected
Default Value:

myDescription - String which contains service of the subscription.
Data Type:RWCString
Privilege:Protected
Default Value:

myDurationType - Time duration of subscriptions (i.e., can be done one time or forever (outstanding).
Data Type:enum EcClSubscriptionType
Privilege:Protected
Default Value:{ONCE, OUTSTANDING}

myExpirationDate - Identifies when this subscription will expire and be removed from the system. The value may be "never" (i.e. the subscription is permanent)
Data Type:RWDate
Privilege:Protected
Default Value:

mySubmittedFlag - Flag which shows whether the subscription has been submitted or not.
Data Type:RWBoolean
Privilege:Protected
Default Value:RWTrue

myUserInfo - Client information, provided by client software.
Data Type:EcClClient
Privilege:Protected
Default Value:

ourCollectorvector - Static vector of pointers to DsClSubscriptionCollector objects, one per dataserver.
Data Type:EcClSubscriptionCollectionVector
Privilege:Protected
Default Value:

Operations:

ClearSubmittedFlag - Flag as not submitted
Arguments:
Return Type:EcTVoid
Privilege:Protected

EcClSubscription - Constructor for client software (therefore, public) which gets attribute information from advertisements, such as service provider. If no collector has been provided, it goes and finds one, based on the static nature of the collector.
Arguments:userinfo, Advertisement&, EcClSubscriptionCollector&
Return Type:Void
Privilege:Public

EcClSubscription - Constructor for already existing collector which gets already existing subscriptions from the sever side through a stream.
Arguments:submittedflag, EcClSubscriptionCollector&, Stream
Return Type:Void
Privilege:Public

GetAction - Means of accessing myAction attribute (object), which will be communicated to the server what this subscription should do when it fires.
Arguments:EcClAction
Return Type:void
Privilege:Public

GetCollector - Get Collector object
Arguments:
Return Type:EcClSubscriptionCollector*
Privilege:Protected

GetDescription - Returns description, containing the service, as a RogueWave string.
Arguments:
Return Type:RWCString
Privilege:Public

GetDurationtype - Accesses myDurationType attribute as to whether subscriptions are done one time or forever (outstanding).
Arguments:
Return Type:EcEClSubscriptionType
Privilege:Public

GetExpirationdate - Public access to myExpirationDate attribute, which provides the expiration date of the subscription.
Arguments:
Return Type:RWDate
Privilege:Public

GetSubmittedflag - Public access to flag as to whether a subscription has been submitted.
Arguments:
Return Type:RWBoolean
Privilege:Public

GetUserinfo - Public access to user information which can be put into the DsSrClient object.
Arguments:GLClient&
Return Type:Void
Privilege:Public

IsSubmitted - Indicate current state
Arguments:
Return Type:RWBoolean
Privilege:Protected

SetAction - Sets the myAction attribute for this particular subscription as determined by the client software.
Arguments:EcClAction&
Return Type:void
Privilege:Public

SetDescription - Allows the client software to fill in the Description attribute with service information.
Arguments:RWCString
Return Type:void
Privilege:Public

SetDurationType - Sets the attribute which determines the existence type of the subscription.
Arguments:EcEClSubscriptionType
Return Type:void
Privilege:Public

SetExpirationDate - Sets the expiration date of the subscription itself.
Arguments:RWDate
Return Type:void
Privilege:Public

SetSubmittedFlag - Sets the flag which indicates whether or not the DsClSubscription has actually been submitted to the dataserver (i.e. the client software is finished with filling in the information, and has invoked the Submit method).
Arguments:RWBoolean
Return Type:void
Privilege:Protected

Submit - Submits subscription to the subscription collector.
Arguments:
Return Type:GlStatus&
Privilege:Public

Update - Replace this subscription in the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

Withdraw - Deletes a subscription from the subscription collector.
Arguments:
Return Type:GlStatus&
Privilege:Public

~EcClSubscription - The DsClSubscription's destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClSubscription class has associations with the following classes:
EcClSubscriptionCollector (Aggregation)

4.5.3.3.6 EcClSubscriptionCollector Class

Parent Class:EcClGenConnector
Public:Yes
Distributed Object:Yes
Purpose and Description:
This public, distributed class is a specialization of the Collector class which handles DsClSubscriptions. This class provides, in addition to the normal vector operations, the ability to create a list of all subscriptions for a given user or advertisement, and a means of submitting and cancelling subscriptions. There are no attributes for this object.

Attributes:

myStatus - This attribute allows the object to maintain information on current status.
Data Type:GlStatus
Privilege:Private
Default Value:

Operations:

BuildList - This operation creates a list of all subscriptions for a given event.
Arguments:Advertisement&
Return Type:const GlStatus &
Privilege:Public

BuildList - This operation creates a list of all subscriptions for a given user.
Arguments:MSS_UserProfile &
Return Type:const GlStatus &
Privilege:Public

BuildListB - This operation allows ops/admin staff to get a list of all subscriptions in the system.
Arguments:
Return Type:const GlStatus &
Privilege:Public

CancelSubscription
Arguments:EcClSubscription&

CreateSubscription - This is a private service used to build the set of subscriptions contained by the SubscriptionCollector.
Arguments:RWBoolean SubmittedFlag, istream &Stream, EcClSubscriptionCollector *me
Return Type:EcClSubscription*
Privilege:Private

CreateSubscription - This function makes a new CsClSubscription with the provided data and adds it to the CsClSubscriptionCollector.
Arguments:GlParameterList&
Return Type:EcUtStatus
Privilege:Protected

EcClSubscriptionCollector - The constructor for DsClSubscriptionCollector's. This constructor establishes a set of Subscriptions for the user based on the provided science data server and the user information.
Arguments:GlUR &dataserver, MSS_UserProfile &
Return Type:Void
Privilege:Public

Populate - This method generates CsClSubscription objects for each of the sets of information in the provided parameter list.
Arguments:GlParameterList&
Return Type:EcUtStatus
Privilege:Protected

SubmitRequest
Arguments:EcClRequest&
Return Type:GlParameterList&
Privilege:Public

SubmitSubscription - This function registers the provided subscription with the subscription server.
Arguments:EcClSubscription&
Return Type:EcTVoid
Privilege:Public

saveOn - This function produces human-readable formatting of the contents of the object such that it can be used in the overloading of the << operator.
Arguments:ostream&
Return Type:EcTVoid
Privilege:Private

~EcClSubscriptionCollector - The EcClSubscriptionCollector's destructor.
Arguments:

Return Type:Void
Privilege:Public

Associations:

The EcClSubscriptionCollector class has associations with the following classes:
EcClCollectorVector (Aggregation)

4.5.3.3.7 EcShActionBase Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:
Base class for actions to be performed when subscriptions fire.

Attributes:

myNotify - Whether to notify on firing
Data Type:RWBoolean
Privilege:Protected
Default Value:

myRequestFlag - Whether request has been set
Data Type:RWBoolean
Privilege:Protected
Default Value:

myText - Notification text
Data Type:RWCString
Privilege:Protected
Default Value:

Operations:

ClearRequest - Clear any request that has been set
Arguments:
Return Type:EcTVoid
Privilege:Public

EcShActionBase
Arguments:RWBoolean, RWCString
Return Type:Void
Privilege:Public

GetNotify - Obtains current value of notify flag
Arguments:
Return Type:RWBoolean
Privilege:Public

GetText - Obtains notification text
Arguments:
Return Type:RWCString
Privilege:Public

HasRequest - Find out if any request has been set
Arguments:
Return Type:RWBoolean
Privilege:Public

SetNotify
Arguments:RWBoolean
Return Type:EcTVoid
Privilege:Public

SetText
Arguments:RWCString
Return Type:EcTVoid
Privilege:Public

~EcShActionBase - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcShActionBase class has associations with the following classes:
None

4.5.3.3.8 EcShSubscription Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myEventID - event subscribed to
Data Type:EcTSbEventID
Privilege:Protected
Default Value:

myExpirationdate - date subscription runs out
Data Type:RWDate
Privilege:Protected
Default Value:

myStartDate - date subsciption was made
Data Type:RWDate
Privilege:Protected
Default Value:

myStatus - Status of this object
Data Type:EcUtStatus
Privilege:Protected
Default Value:

myUserInfo - subscriber information
Data Type:MsAcUserProfile
Privilege:Protected
Default Value:

Operations:

EcShSubscription - normal constructor
Arguments:MsAcUserProfile
Return Type:Void
Privilege:Public

EcShSubscription - copy constructor
Arguments:const
Return Type:Void
Privilege:Public

EcShSubscription - default constructor defined here to prevent anyone from using it
Arguments:
Return Type:Void
Privilege:Public

GetEventID - get value of myUserInfo
Arguments:

Return Type:EcTSbEventID&
Privilege:Public

GetExpirationDate - get value of myUserInfo
Arguments:
Return Type:RWDate
Privilege:Public

GetStartDate - get value of myUserInfo
Arguments:
Return Type:RWDate
Privilege:Public

GetUserInfo - get value of myUserInfo
Arguments:
Return Type:MsAcUserProfile&
Privilege:Public

Getstatus - get status of this instance
Arguments:
Return Type:EcUtStatus
Privilege:Public

SetEventID - set myUserInfo attribute to provided info
Arguments:EcTSbEventID&
Return Type:void
Privilege:Public

SetExpirationDate - set myUserInfo attribute to provided info
Arguments:RWDate expirationDate
Return Type:void
Privilege:Public

SetStartDate - set myUserInfo attribute to provided info
Arguments:RWDate startDate
Return Type:void
Privilege:Public

SetUserInfo - set myUserInfoattribute to
Arguments:MsAcUserProfile&
Return Type:void
Privilege:Public

saveOn - Debugging aid Input/Output. send msgs to
Arguments:ostream&

Return Type:void
Privilege:Public

~EcShSubscription - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcShSubscription class has associations with the following classes:
None

4.5.3.4 Object Model, Part 2

See Figure 4.5.3.4-1.

4.5.3.4.1 EcDbEventStore Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

None

Operations:

Add -
Arguments:EcSbEvent&
Return Type:EcUtStatus &
Privilege:Public

GetEvents -
Arguments:EcSbEventHandler&
Return Type:EcUtStatus &
Privilege:Public

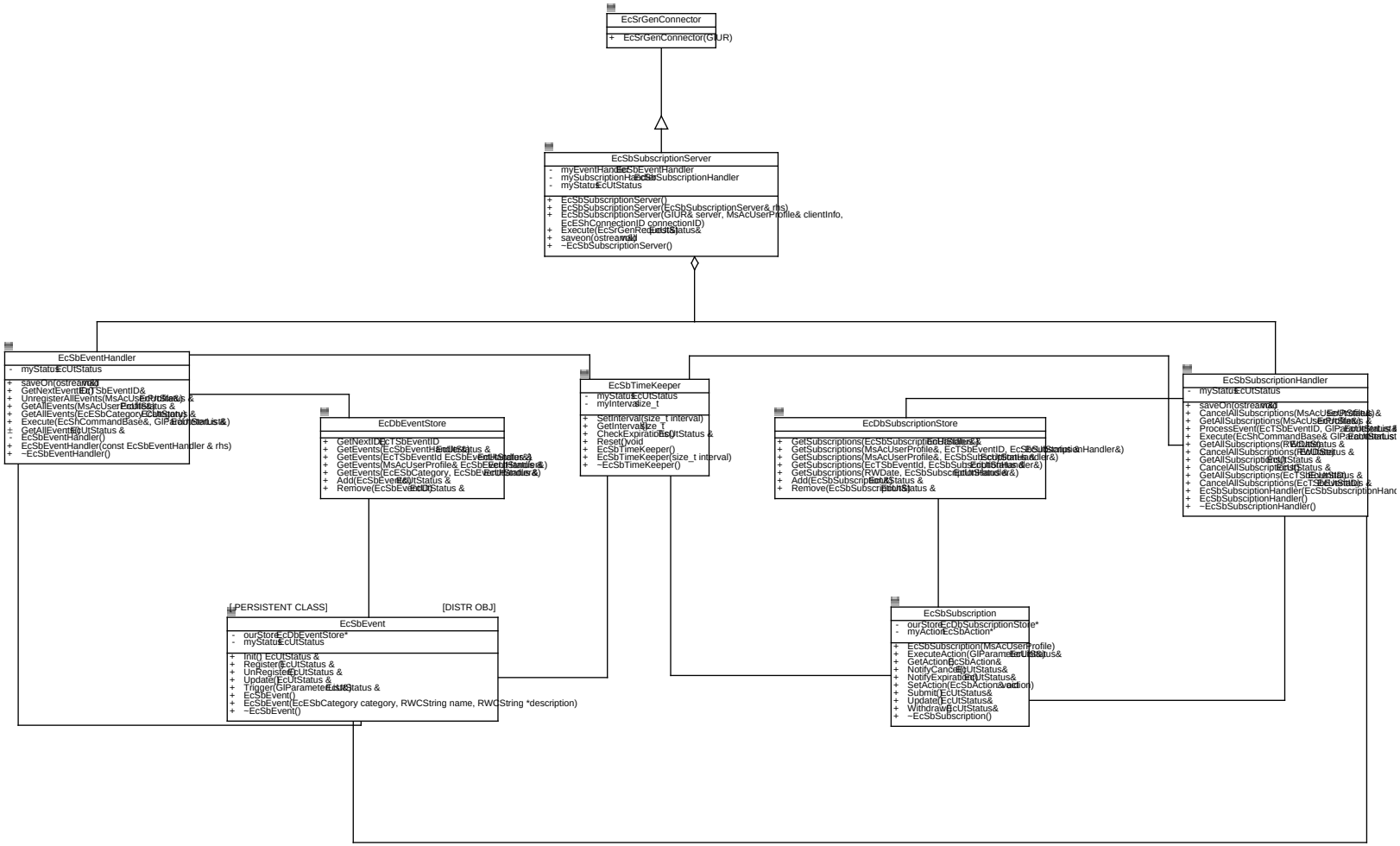

Figure 4.5.3.4-1. Subscription Server Object Model, Part 2

GetEvents -
Arguments:EcTSbEventId EcSbEventHandler&
Return Type:EcUtStatus &
Privilege:Public

GetEvents -
Arguments:MsAcUserProfile& EcSbEventHandler&
Return Type:EcUtStatus &
Privilege:Public

GetEvents -
Arguments:EcESbCategory, EcSbEventHandler&
Return Type:EcUtStatus &
Privilege:Public

GetNextID -
Arguments:
Return Type:EcTSbEventID
Privilege:Public

Remove -
Arguments:EcSbEventID
Return Type:EcUtStatus &
Privilege:Public

Associations:

The EcDbEventStore class has associations with the following classes:
Class: EcSbEvent
Class: EcSbEventHandler

4.5.3.4.2 EcDbSubscriptionStore Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

None

Operations:

Add -
Arguments:EcSbSubscription&
Return Type:EcUtStatus &
Privilege:Public

GetSubscriptions -
Arguments:EcSbSubscriptionHandler&
Return Type:EcUtStatus &
Privilege:Public

GetSubscriptions -
Arguments:MsAcUserProfile&, EcTSbEventID, EcSbSubscriptionHandler&
Return Type:EcUtStatus &
Privilege:Public

GetSubscriptions -
Arguments:MsAcUserProfile&, EcSbSubscriptionHandler&
Return Type:EcUtStatus &
Privilege:Public

GetSubscriptions -
Arguments:EcTSbEventId, EcSbSubscriptionHandler&
Return Type:EcUtStatus &
Privilege:Public

GetSubscriptions -
Arguments:RWDate, EcSbSubscriptionHandler&
Return Type:EcUtStatus &
Privilege:Public

Remove -
Arguments:EcSbSubscription&
Return Type:EcUtStatus &
Privilege:Public

Associations:

The EcDbSubscriptionStore class has associations with the following classes:
Class: EcSbSubscription
Class: EcSbSubscriptionHandler

4.5.3.4.3 EcSbEvent Class

Parent Class:Not Applicable
Public:No

Distributed Object:Yes
Persistent Class:True
Purpose and Description:
Defines events used by the server.

Attributes:

myStatus -
Data Type:EcUtStatus
Privilege:Private
Default Value:

ourStore -
Data Type:EcDbEventStore*
Privilege:Private
Default Value:

Operations:

EcSbEvent - default constructor
Arguments:
Return Type:Void
Privilege:Public

EcSbEvent - aut0o-fill constructor
Arguments:EcESbCategory category, RWCString name, RWCString *description
Return Type:Void
Privilege:Public

Init - do set up work such as initialize the event store
Arguments:
Return Type:EcUtStatus &
Privilege:Public

Register - register to the server as being able to occur
Arguments:
Return Type:EcUtStatus &
Privilege:Public

Trigger - tell server that this event occurred
Arguments:GlParameterList&
Return Type:EcUtStatus &
Privilege:Public

UnRegister - unregister from the server
Arguments:
Return Type:EcUtStatus &
Privilege:Public

Update - change values
Arguments:
Return Type:EcUtStatus &
Privilege:Public

~EcSbEvent - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbEvent class has associations with the following classes:
Class: EcDbEventStore
Class: EcSbEventHandler
Class: EcSbSubscriptionHandler
Class: EcSbTimeKeeper

4.5.3.4.4 EcSbEventHandler Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myStatus -
Data Type:EcUtStatus
Privilege:Private
Default Value:

Operations:

EcSbEventHandler - default constructor
Arguments:
Return Type:Void
Privilege:Private

EcSbEventHandler - copy constructor
Arguments:const EcSbEventHandler & rhs
Return Type:Void
Privilege:Public

Execute - interpret and execute this command
Arguments:EcShCommandBase&, GlParameterList&
Return Type:EcUtStatus &
Privilege:Public

GetAllEvents - get all events created by this entry
Arguments:MsAcUserProfile&
Return Type:EcUtStatus &
Privilege:Public

GetAllEvents - get all events in this category
Arguments:EcESbCategory Category
Return Type:EcUtStatus &
Privilege:Public

GetAllEvents - get all events in system(by operator)
Arguments:
Return Type:EcUtStatus &
Privilege:Protected

GetNextEventID - provide ID to new event
Arguments:
Return Type:EcTSbEventID&
Privilege:Public

UnregisterAllEvents - remove all events created by this entry
Arguments:MsAcUserProfile&
Return Type:EcUtStatus &
Privilege:Public

saveOn - hyman readable stream on which to write
Arguments:ostream&
Return Type:void
Privilege:Public

~EcSbEventHandler - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbEventHandler class has associations with the following classes:
Class: EcDbEventStore
Class: EcSbEvent
Class: EcSbTimeKeeper
EcSbSubscriptionServer (Aggregation)

4.5.3.4.5 EcSbSubscription Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Persistent Class:
Purpose and Description:
Records all characteristics of a single subscription. It is responsible for executing the requested action when the associated event occurs.

Attributes:

myAction -
Data Type:EcSbAction*
Privilege:Private
Default Value:

ourStore -
Data Type:EcDbSubscriptionStore*
Privilege:Private
Default Value:

Operations:

EcSbSubscription - normal constructor
Arguments:MsAcUserProfile
Return Type:Void
Privilege:Public

ExecuteAction - perform the subscriber's action
Arguments:GlParameterList&
Return Type:EcUtStatus&
Privilege:Public

GetAction - get this objects action
Arguments:

Return Type:EcSbAction&
Privilege:Public

NotifyCancel - warn subscriber about cancelation
Arguments:
Return Type:EcUtStatus&
Privilege:Public

NotifyExpiration - warn subscriber about end date
Arguments:
Return Type:EcUtStatus&
Privilege:Public

SetAction - get this objects action
Arguments:EcSbAction& action
Return Type:void
Privilege:Public

Submit - register this subscription
Arguments:
Return Type:EcUtStatus&
Privilege:Public

Update - replace this subscription
Arguments:
Return Type:EcUtStatus&
Privilege:Public

Withdraw - cancel this subscription
Arguments:
Return Type:EcUtStatus&
Privilege:Public

~EcSbSubscription - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbSubscription class has associations with the following classes:
Class: EcDbSubscriptionStore
Class: EcSbSubscriptionHandler
Class: EcSbTimeKeeper

4.5.3.4.6 EcSbSubscriptionHandler Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myStatus -
Data Type:EcUtStatus
Privilege:Private
Default Value:

Operations:

CancelAllSubscriptions - remove all subscriptions for this entity
Arguments:MsAcUserProfile&
Return Type:EcUtStatus &
Privilege:Public

CancelAllSubscriptions - cancell all subscription for this date
Arguments:RWDate
Return Type:EcUtStatus &
Privilege:Public

CancelAllSubscriptions - cancel all subscription for the operators
Arguments:
Return Type:EcUtStatus &
Privilege:Public

CancelAllSubscriptions - remove all subscriptions for this event
Arguments:EcTSbEventID
Return Type:EcUtStatus &
Privilege:Public

EcSbSubsciptionHandler - copy constructor
Arguments:EcSbSubscriptionHandler
Return Type:Void
Privilege:Public

EcSbSubsciptionHandler -
Arguments:
Return Type:Void

Privilege:Public

Execute - interpret and execute this command with these parameters
Arguments:EcShCommandBase& GlParameterList&
Return Type:EcUtStatus &
Privilege:Public

GetAllSubscriptions - find all subscriptions for this entity
Arguments:MsAcUserProfile&
Return Type:EcUtStatus &
Privilege:Public

GetAllSubscriptions - find all subscriptions for this date
Arguments:RWDate
Return Type:EcUtStatus &
Privilege:Public

GetAllSubscriptions - get all subscriptions in the systems
Arguments:
Return Type:EcUtStatus &
Privilege:Public

GetAllSubscriptions - get all subscription for this event
Arguments:EcTSbEventID
Return Type:EcUtStatus &
Privilege:Public

ProcessEvent - fire subscription for this event
Arguments:EcTSbEventID, GlParameterList&
Return Type:EcUtStatus &
Privilege:Public

saveOn - human readable stream on which to write
Arguments:ostream&
Return Type:void
Privilege:Public

~EcSbSubscriptionHandler - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbSubscriptionHandler class has associations with the following classes:

Class: EcDbSubscriptionStore
Class: EcSbEvent
Class: EcSbSubscription
Class: EcSbTimeKeeper
EcSbSubscriptionServer (Aggregation)

4.5.3.4.7 EcSbSubscriptionServer Class

Parent Class:EcSrGenConnector
Public:No
Distributed Object:No
Purpose and Description:
Provides an interface to subscription services. Other objects can use it to access all administrative functions that are necessary for creating, removing, and managing subscriptions. It also handles event registration, and returns a handle to the DsSbEventHandler to registering events.

Attributes:

myEventHandler -
Data Type:EcSbEventHandler
Privilege:Private
Default Value:

myStatus -
Data Type:EcUtStatus
Privilege:Private
Default Value:

mySubscriptionHandler -
Data Type:EcSbSubscriptionHandler
Privilege:Private
Default Value:

Operations:

EcSbSubscriptionServer - default constructor
Arguments:
Return Type:Void
Privilege:Public

EcSbSubscriptionServer - copy constructor
Arguments:EcSbSubscriptionServer& rhs

Return Type:Void
Privilege:Public

EcSbSubscriptionServer - constructor : data server to use user info specific connection
Arguments:GIUR& server, MsAcUserProfile& clientInfo, EcEShConnectionID connectionID
Return Type:Void
Privilege:Public

Execute - execute this request
Arguments:EcSrGenRequest&
Return Type:EcUtStatus&
Privilege:Public

saveon - human readable stream on which to write
Arguments:ostream&
Return Type:void
Privilege:Public

~EcSbSubscriptionServer - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbSubscriptionServer class has associations with the following classes:
None

4.5.3.4.8 EcSbTimeKeeper Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myInterval -
Data Type:size_t
Privilege:Private
Default Value:

myStatus -

Data Type:EcUtStatus
Privilege:Private
Default Value:

Operations:

CheckExpirations - notify/cancel
Arguments:
Return Type:EcUtStatus &
Privilege:Public

EcSbTimeKeeper - default constructor
Arguments:
Return Type:Void
Privilege:Public

EcSbTimeKeeper - normal constructor
Arguments:size_t interval
Return Type:Void
Privilege:Public

GetInterval - get value for my interval
Arguments:
Return Type:size_t
Privilege:Public

Reset - go back to sleep
Arguments:
Return Type:void
Privilege:Public

SetInterval - set new value for myInterval amount of time
Arguments:size_t interval
Return Type:Void
Privilege:Public

~EcSbTimeKeeper - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbTimeKeeper class has associations with the following classes:
Class: EcSbEvent
Class: EcSbEventHandler
Class: EcSbSubscription
Class: EcSbSubscriptionHandler

4.5.3.4.9 EcSrGenConnector Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

None

Operations:

EcSrGenConnector -
Arguments:GIUR
Return Type:Void
Privilege:Public

Associations:

The EcSrGenConnector class has associations with the following classes:
None

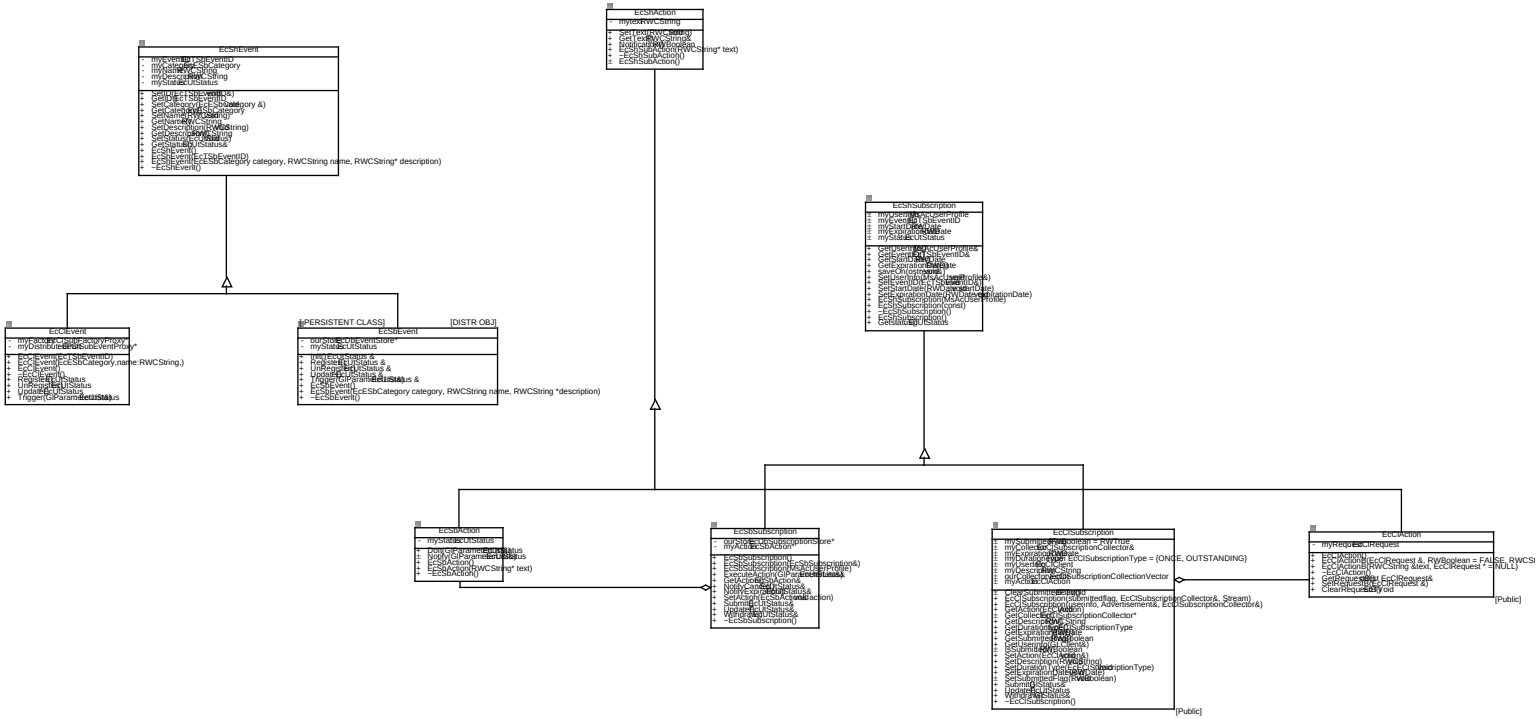

Figure 4.5.3.5-1. Subscription Server Object Model, Part 3

4.5.3.5.1 EcClAction Class

Parent Class:EcShAction
Public:Yes
Distributed Object:No
Purpose and Description:
A client interface object that represents the components of the action to be performed when a subscription is triggered. The possibilities are that the client will receive a notification (including all parameters that are returned by the object that triggers the subscription and an optional piece of client-specified text) and/or a request that will be executed. The client is required to specify an action for each subscription.

Attributes:

myRequest - The request that is currently associated with this action.
Data Type:EcClRequest
Privilege:Private
Default Value:

Operations:

ClearRequestB - Used to clear any request that has been set for the action.
Arguments:
Return Type:EcTVoid
Privilege:Public

EcCIAction - Default constructor
Arguments:
Return Type:Void
Privilege:Public

EcClActionB - Used to construct an action from a piece of text and, RELEASE B: optionally, a request. The notification flag is set.
Arguments:RWCString &text, EcClRequest * = NULL
Return Type:Void
Privilege:Public

EcClActionB - Used to construct an action from a request and, optionally, a value for the notification flag and a piece of text.
Arguments:EcClRequest &, RWBoolean = FALSE, RWCString * = NULL
Return Type:Void
Privilege:Public

GetRequestB - Returns the request currently set for the action.
Arguments:
Return Type:const EcClRequest&
Privilege:Public

SetRequestB - Sets the request to be executed when the subscription fires.
Arguments:EcClRequest &
Return Type:Void
Privilege:Public

~EcClAction - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClAction class has associations with the following classes:
EcClSubscription (Aggregation)

4.5.3.5.2 EcClEvent Class

Parent Class:EcShEvent
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myDistributedPart - The distributed part of CsClEvent
Data Type:EcClSubEventProxy*
Privilege:Private
Default Value:

myFactory - This is a pointer to the subscription server's factory distributed object. This provides client interface to the subscription server's factory distributed object, and is used to create client connection and request objects.
Data Type:EcClSubFactoryProxy*
Privilege:Private
Default Value:

Operations:

EcClEvent - full constructor, creates auto-filled-in new event object
Arguments:EcESbCategory,name:RWCString,
Return Type:Void
Privilege:Public

EcClEvent - Usual constructor, takes event ID and instantiates the event
Arguments:EcTSbEventID
Return Type:Void
Privilege:Public

EcClEvent - Defualt constructor
Arguments:
Return Type:Void
Privilege:Public

Register - Put this event into the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

Trigger - Activate this event in the system such that all subscriptions to this event will be processed
Arguments:GlParameterList&
Return Type:EcUtStatus
Privilege:Public

UnRegister - Take this event out of the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

Update - Change this event's value in the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

~EcClEvent - Destructor, cleans up and removes this object
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClEvent class has associations with the following classes:
None

4.5.3.5.3 EcClSubscription Class

Parent Class:EcShSubscription
Public:Yes
Distributed Object:No
Purpose and Description:
This class is the client side subscription which can either be created from advertisements or from exisiting subscriptions from the server side (through a stream.)

Attributes:

myAction - The action to be performed when the subscription fires.
Data Type:EcClAction
Privilege:Protected
Default Value:

myCollector - A pointer to the collector that this reference is a member of. If this pointer is null, then this reference is a member of one of the collectors in the static collector vector.
Data Type:EcClSubscriptionCollector&
Privilege:Protected
Default Value:

myDescription - String which contains service of the subscription.
Data Type:RWCString
Privilege:Protected
Default Value:

myDurationType - Time duration of subscriptions (i.e., can be done one time or forever (outstanding).
Data Type:enum EcClSubscriptionType
Privilege:Protected
Default Value:{ONCE, OUTSTANDING}

myExpirationDate - Identifies when this subscription will expire and be removed from the system. The value may be "never" (i.e. the subscription is permanent)
Data Type:RWDate
Privilege:Protected
Default Value:

mySubmittedFlag - Flag which shows whether the subscription has been submitted or not.
Data Type:RWBoolean
Privilege:Protected

Default Value:RWTrue

myUserInfo - Client information, provided by client software.
Data Type:EcClClient
Privilege:Protected
Default Value:

ourCollectorvector - Static vector of pointers to DsClSubscriptionCollector objects, one per dataserver.
Data Type:EcClSubscriptionCollectionVector
Privilege:Protected
Default Value:

Operations:

ClearSubmittedFlag - Flag as not submitted
Arguments:
Return Type:EcTVoid
Privilege:Protected

EcClSubscription - Constructor for client software (therefore, public) which gets attribute information from advertisements, such as service provider. If no collector has been provided, it goes and finds one, based on the static nature of the collector.
Arguments:userinfo, Advertisement&, EcClSubscriptionCollector&
Return Type:Void
Privilege:Public

EcClSubscription - Constructor for already existing collector which gets already existing subscriptions from the sever side through a stream.
Arguments:submittedflag, EcClSubscriptionCollector&, Stream
Return Type:Void
Privilege:Public

GetAction - Means of accessing myAction attribute (object), which will be communicated to the server what this subscription should do when it fires.
Arguments:EcClAction
Return Type:void
Privilege:Public

GetCollector - Get Collector object
Arguments:
Return Type:EcClSubscriptionCollector*
Privilege:Protected

GetDescription - Returns description, containing the service, as a RogueWave string.
Arguments:
Return Type:RWCString
Privilege:Public

GetDurationtype - Accesses myDurationType attribute as to whether subscriptions are done one time or forever (outstanding).
Arguments:
Return Type:EcEClSubscriptionType
Privilege:Public

GetExpirationdate - Public access to myExpirationDate attribute, which provides the expiration date of the subscription.
Arguments:
Return Type:RWDate
Privilege:Public

GetSubmittedflag - Public access to flag as to whether a subscription has been submitted.
Arguments:
Return Type:RWBoolean
Privilege:Public

GetUserinfo - Public access to user information which can be put into the DsSrClient object.
Arguments:GLClient&
Return Type:Void
Privilege:Public

IsSubmitted - Indicate current state
Arguments:
Return Type:RWBoolean
Privilege:Protected

SetAction - Sets the myAction attribute for this particular subscription as determined by the client software.
Arguments:EcClAction&
Return Type:void
Privilege:Public

SetDescription - Allows the client software to fill in the Description attribute with service information.
Arguments:RWCString
Return Type:void
Privilege:Public

SetDurationType - Sets the attribute which determines the existence type of the subscription.
Arguments:EcEClSubscriptionType
Return Type:void
Privilege:Public

SetExpirationDate - Sets the expiration date of the subscription itself.
Arguments:RWDate
Return Type:void
Privilege:Public

SetSubmittedFlag - Sets the flag which indicates whether or not the DsClSubscription has actually been submitted to the dataserver (i.e. the client software is finished with filling in the information, and has invoked the Submit method).
Arguments:RWBoolean
Return Type:void
Privilege:Protected

Submit - Submits subscription to the subscription collector.
Arguments:
Return Type:GlStatus&
Privilege:Public

Update - Replace this subscription in the system
Arguments:
Return Type:EcUtStatus
Privilege:Public

Withdraw - Deletes a subscription from the subscription collector.
Arguments:
Return Type:GlStatus&
Privilege:Public

~EcClSubscription - The DsClSubscription's destructor.
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcClSubscription class has associations with the following classes:
None

4.5.3.5.4 EcSbAction Class

Parent Class:EcShAction
Public:No
Distributed Object:No
Purpose and Description:
This defines an activity to be performed on behalf of a server when a previously defined and advertised event occurs. Currently, notifications and requests are valid actions.

Attributes:

myStatus
Data Type:EcUtStatus
Privilege:Private
Default Value:

Operations:

DoIt - do whatever action is specified
Arguments:GlParameterList&
Return Type:EcUtStatus
Privilege:Public

EcSbAction - default constructor
Arguments:
Return Type:Void
Privilege:Public

EcSbAction - text only constructor
Arguments:RWCString* text
Return Type:Void
Privilege:Public

Notify - send the notification text with the supplied parameters
Arguments:GlParameterList&
Return Type:EcUtStatus
Privilege:Protected

~EcSbAction - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbAction class has associations with the following classes:
EcSbSubscription (Aggregation)

4.5.3.5.5 EcSbEvent Class

Parent Class:EcShEvent
Public:No
Distributed Object:Yes
Persistent Class:True
Purpose and Description:
Defines events used by the server.

Attributes:

myStatus -
Data Type:EcUtStatus
Privilege:Private
Default Value:

ourStore -
Data Type:EcDbEventStore*
Privilege:Private
Default Value:

Operations:

EcSbEvent - default constructor
Arguments:
Return Type:Void
Privilege:Public

EcSbEvent - auto-fill constructor
Arguments:EcESbCategory category, RWCString name, RWCString *description
Return Type:Void
Privilege:Public

Init - do set up work such as initialize the event store
Arguments:
Return Type:EcUtStatus &
Privilege:Public

Register - register to the server as being able to occur
Arguments:
Return Type:EcUtStatus &
Privilege:Public

Trigger - tell server that this event occurred
Arguments:GlParameterList&
Return Type:EcUtStatus &
Privilege:Public

UnRegister - unregister from the server
Arguments:
Return Type:EcUtStatus &
Privilege:Public

Update - change values
Arguments:
Return Type:EcUtStatus &
Privilege:Public

~EcSbEvent - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbEvent class has associations with the following classes:
None

4.5.3.5.6 EcSbSubscription Class

Parent Class:EcShSubscription
Public:No
Distributed Object:No
Persistent Class:
Purpose and Description:
Records all characteristics of a single subscription. It is responsible for executing the requested action when the associated event occurs.

Attributes:

myAction -
Data Type:EcSbAction*

Privilege:Private
Default Value:

ourStore -
Data Type:EcDbSubscriptionStore*
Privilege:Private
Default Value:

Operations:

EcSbSubscription - normal constructor
Arguments:MsAcUserProfile
Return Type:Void
Privilege:Public

EcSbSubscription - copy constructor
Arguments:EcSbSubscription&
Return Type:Void
Privilege:Public

EcSbSubscription - default constructor defined here to prevent anyone from using it
Arguments:
Return Type:Void
Privilege:Public

ExecuteAction - perform the subscriber's action
Arguments:GlParameterList&
Return Type:EcUtStatus&
Privilege:Public

GetAction - get this objects action
Arguments:
Return Type:EcSbAction&
Privilege:Public

NotifyCancel - warn subscriber about cancelation
Arguments:
Return Type:EcUtStatus&
Privilege:Public

NotifyExpiration - warn subscriber about end date
Arguments:
Return Type:EcUtStatus&
Privilege:Public

SetAction - get this objects action
Arguments:EcSbAction& action
Return Type:void
Privilege:Public

Submit - register this subscription
Arguments:
Return Type:EcUtStatus&
Privilege:Public

Update - replace this subscription
Arguments:
Return Type:EcUtStatus&
Privilege:Public

Withdraw - cancel this subscription
Arguments:
Return Type:EcUtStatus&
Privilege:Public

~EcSbSubscription - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbSubscription class has associations with the following classes:
None

4.5.3.5.7 EcShAction Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

mytext
Data Type:RWCString
Privilege:Private

Default Value:

Operations:

EcShSubAction - normal constructor
Arguments:RWCString* text
Return Type:Void
Privilege:Public

EcShSubAction - hidden default cinstructor
Arguments:
Return Type:Void
Privilege:Protected

GetText - obtain notification text
Arguments:
Return Type:RWCString&
Privilege:Public

Notification - obtain current value of notify flag
Arguments:
Return Type:RWBoolean
Privilege:Public

SetText - set notification text
Arguments:RWCString
Return Type:void
Privilege:Public

~EcShSubAction - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcShAction class has associations with the following classes:
None

4.5.3.5.8 EcShEvent Class

Parent Class:Not Applicable
Public:No
Distributed Object:No

Purpose and Description:

Attributes:

myCategory
Data Type:EcESbCategory
Privilege:Private
Default Value:

myDescription
Data Type:RWCString
Privilege:Private
Default Value:

myEventID
Data Type:EcTSbEventID
Privilege:Private
Default Value:

myName
Data Type:RWCString
Privilege:Private
Default Value:

myStatus
Data Type:EcUtStatus
Privilege:Private
Default Value:

Operations:

EcShEvent - default constructor: create new empty and fill
Arguments:
Return Type:Void
Privilege:Public

EcShEvent - constructor uses existing eventid
Arguments:EcTSbEventID
Return Type:Void
Privilege:Public

EcShEvent - constructor for new auto-filled
Arguments:EcESbCategory category, RWCString name, RWCString* description
Return Type:Void

Privilege:Public

GetCategory - provides access to the attribute myCategory
Arguments:
Return Type:EcESbCategory
Privilege:Public

GetDescription - provide access to the attribute myDescription
Arguments:
Return Type:RWCString
Privilege:Public

GetID - provide access to the attribute myEventID
Arguments:
Return Type:EcTSbEventID
Privilege:Public

GetName - provides access to the attribute myName
Arguments:
Return Type:RWCString
Privilege:Public

GetStatus - provide access to the attribute myStatus
Arguments:
Return Type:EcUtStatus&
Privilege:Public

SetCategory - set the attribute myCategory to the specified value
Arguments:EcESbCategory &
Return Type:void
Privilege:Public

SetDescription - sets the attribute myDescription to the specified value
Arguments:RWCString
Return Type:void
Privilege:Public

SetID
Arguments:EcTSbEventID&
Return Type:void
Privilege:Public

SetName - sets the attribute name to the specified value
Arguments:RWCString
Return Type:void

Privilege:Public

SetStatus - sets the attribute myStatus to the specified value
Arguments:EcUtStatus
Return Type:void
Privilege:Public

~EcShEvent - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcShEvent class has associations with the following classes:
None

4.5.3.5.9 EcShSubscription Class

Parent Class:Not Applicable
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

myEventID - event subscribed to
Data Type:EcTSbEventID
Privilege:Protected
Default Value:

myExpirationdate - date subscription runs out
Data Type:RWDate
Privilege:Protected
Default Value:

myStartDate - date subsciption was made
Data Type:RWDate
Privilege:Protected
Default Value:

myStatus - Status of this object
Data Type:EcUtStatus
Privilege:Protected

Default Value:

myUserInfo - subscriber information
Data Type:MsAcUserProfile
Privilege:Protected
Default Value:

Operations:

EcShSubscription - normal constructor
Arguments:MsAcUserProfile
Return Type:Void
Privilege:Public

EcShSubscription - copy constructor
Arguments:const
Return Type:Void
Privilege:Public

EcShSubscription - default constructor defined here to prevent anyone from using it
Arguments:
Return Type:Void
Privilege:Public

GetEventID - get value of myUserInfo
Arguments:
Return Type:EcTSbEventID&
Privilege:Public

GetExpirationDate - get value of myUserInfo
Arguments:
Return Type:RWDate
Privilege:Public

GetStartDate - get value of myUserInfo
Arguments:
Return Type:RWDate
Privilege:Public

GetUserInfo - get value of myUserInfo
Arguments:
Return Type:MsAcUserProfile&
Privilege:Public

Getstatus - get status of this instance
Arguments:
Return Type:EcUtStatus
Privilege:Public

SetEventID - set myUserInfo attribute to provided info
Arguments:EcTSbEventID&
Return Type:void
Privilege:Public

SetExpirationDate - set myUserInfo attribute to provided info
Arguments:RWDate expirationDate
Return Type:void
Privilege:Public

SetStartDate - set myUserInfo attribute to provided info
Arguments:RWDate startDate
Return Type:void
Privilege:Public

SetUserInfo - set myUserInfoattribute to
Arguments:MsAcUserProfile&
Return Type:void
Privilege:Public

saveOn - Debugging aid Input/Output. send msgs to
Arguments:ostream&
Return Type:void
Privilege:Public

~EcShSubscription - destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcShSubscription class has associations with the following classes:
None

4.5.3.6 Object Model, Part 4

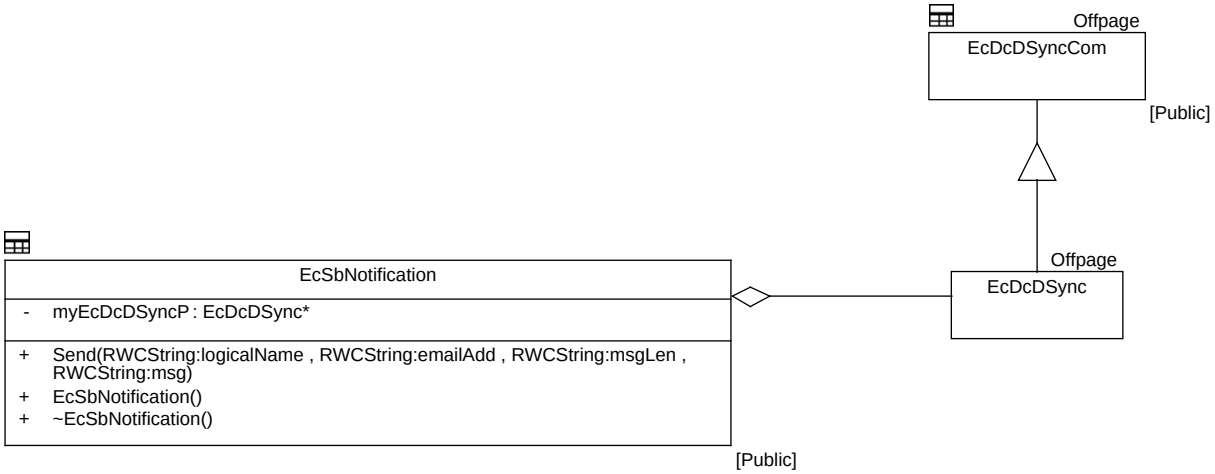

Figure 4.5.3.6-1. Subscription Server Object Model, Part 4

4.5.3.6.1 EcDcDSync Class

Parent Class:EcDcDSyncCom
Public:No
Distributed Object:No
Purpose and Description:

Attributes:

All Attributes inherited from parent class

Operations:

All Operations inherited from parent class

Associations:

The EcDcDSync class has associations with the following classes:
EcSbNotification (Aggregation)

4.5.3.6.2 EcDcDSyncCom Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:

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.

Attributes:

None

Operations:

None

Associations:

The EcDcDSyncCom class has associations with the following classes:
None

4.5.3.6.3 EcSbNotification Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:

Attributes:

myEcDcDSyncP - A pointer to EcDsDSync class
Data Type:EcDcDSync*
Privilege:Private
Default Value:

Operations:

EcSbNotification - This is the default constructor
Arguments:
Return Type:Void
Privilege:Public

Send - This operation will be used to send notification message and its corresponding information.
Arguments:RWCString:logicalName , RWCString:emailAdd , RWCString:msgLen , RWCString:msg
Return Type:Void

Privilege:Public

~EcSbNotification - Destructor
Arguments:
Return Type:Void
Privilege:Public

Associations:

The EcSbNotification class has associations with the following classes:
None

4.5.3.7 Functional Model

The subscription service provides a set of classes with distinct functionality which makes the incorporation of generic subscription functionality into ECS client/server applications easy. The classes can be categorized into three distinct catogories:

(1) classes that provide the basic abstractions of event, action and subscription

(2) classes that provide distributed object behavior, and

(3) classes that basically conduct timekeeping and handling functions

The framework facilitates the following functionality:

- Ability to register events through the use of the event classes
- Ability to create and submit subscriptions through the use of the subscription classes

The class EcSbSubscriptionServer provides the specialized connection for the subscription server on the server-side and also the access to the processing of events and subscriptions. The class EcSbSubscriptionCollector provides the primary interface for submitting subscriptions on the client side. The class EcSbEvent defines the framework for creating an event. The class EcSbSubscription defines the framework for creating a subscription. The class EcSbAction is the basic framework class for creating actions.

Table 4.5.3.7-1 describes the subscription classes.

Table 4.5.3.7-1. Subscription Server Object Responsibility Matrix (1 of 2)

Class name	Description
EcSbSubscriptionServer	Provides the specialized connection for the subscription server and also (through containment) access to the processing of events and subscriptions
EcSbSubscriptionCollector	Provides the primary interface for submitting subscriptions on the client side
EcSbTimeKeeper	Keeps track of time as the subscription server runs, and makes sure that anything linked to time (such as the expiration of subscriptions) occurs. Supports the concept of "Timed Subscriptions"

Table 4.5.3.7-1. Subscription Server Object Responsibility Matrix (2 of 2)

Class name	Description
EcSbEventHandler	Provides the interface to work with events on the server side (server-side management of events)
EcSbSubscriptionHandler	Provides the interface to work with subscriptions on the server side
EcDbEventStore	Provides the interface to interact with a database of events
EcDbSubscriptionStore	Provides the interface to interact with a database of subscriptions
EcSbEvent	Provides the data and behavior that is particular to events on the server i.e. the actual implementation of the event
EcSbAction	Provides the data and behavior that is particular to the server, with respect to what action to perform when a given subscription fires
EcSbSubscription	Provides the data and behavior that is particular to the subscriptions on the server i.e. the actual implementation of the subscriptions
EcClEvent	Provides the data and behavior that is particular to events on the client side
EcClAction	Provides the data and behavior that is particular to actions on the client side
EcClSubscription	Provides the data and behavior that is particular to subscriptions on the client side
EcShEvent	Provides the data and behavior that is common to events on both the client and the server side
EcShAction	Base class for actions to be performed when subscriptions fire
EcShSubscription	Provides the data and behavior that is common to subscriptions on the client and the server side
EcSbNotification	Provides the ability to alert or inform clients of events. This object transmits data from a notification source (e.g. subscription that has been triggered) to a notification receiver (e.g. a user who created a subscription). It attempts to notify the receipient directly, else via e-mail.

4.5.3.8 Dynamic Model Scenarios

4.5.3.8.1 Scenario #1

4.5.3.8.1.1 Abstract

This scenario demonstrates the steps that the client application (CallingObject) must perform to submit a subscription on behalf of an end-user. In this scenario the client has retrieved a subscription advertisement from the Advertising server and has decided that he/she would like to be notified upon the future occurrence of that advertised event.

4.5.3.8.1.1 Interfaces

The client object and the Advertising server

4.5.3.8.1.2 Stimulus

The client application constructs the EcClSubscription object with the subscription information from the advertisement.

4.5.3.8.1.3 Desired Response

The subscription is added to the subscription list maintained by the appropriate subscription server.

4.5.3.8.1.4 Participating Classes

EcClSubscription

EcClSubscriptionCollector

EcSbSubscriptionServer

EcSbSubscriptionHandler

4.5.3.8.1.5 Pre-conditions

The preconditions for this scenario are that a user has selected an advertised subscription from the Advertiser (that represents the event that he/she is interested in) and the client application has created an instance of the advertisement. The advertisement contains information about the event that will trigger the subscription. The user may specify an action for the data server to perform whenever the subscribed event occurs. For example, the user could request automatic distribution whenever new CERES02 data arrives.

4.5.3.8.1.6 Post-conditions

The subscription is activated.

4.5.3.8.1.7 Scenario Description

1. The scenario primitive starts with the client application creating a subscription, passing the advertisement (event information) and user information (notification recipient).
2. The client then creates an action which includes a request and an optional notification text string (user-entered data).
3. Next the EcClSubscription object's SetAction() operation is used to register the user defined action.
4. Finally, the server uses AddSubscriptions to add the new subscription to its list of managed subscriptions.

4.5.3.8.1.8 Event Trace

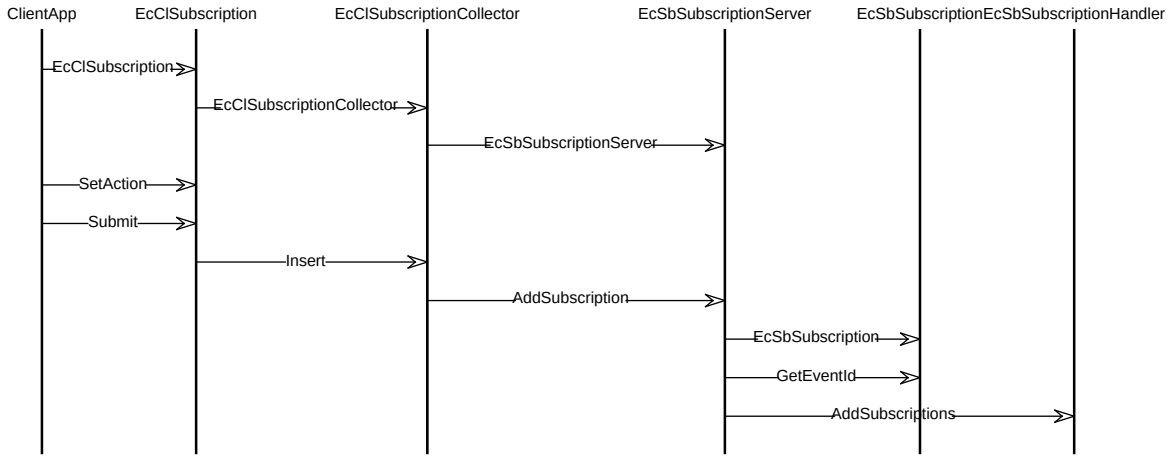

Figure 4.5.3.8-1 Event Trace for Submitting a Subscription

4.5.3.8.2 Scenario #2

4.5.3.8.2.1 Abstract

This scenario shows the steps that are executed when a subscription expires. Note that the user is notified in advance of a subscription expiration, and it only expires if it's not renewed.

4.5.3.8.2.2 Interfaces

No external clients are involved in this scenerio. Internally, the EcSbSubscriptionServer object is being used.

4.5.3.8.2.3 Stimulus

There is one subscription that has expired today and the subscription server callback timer fires.

4.5.3.8.2.4 Desired Response

The subscription is removed from the subscription list.

4.5.3.8.2.5 Participating Classes

EcSbCallbackTimer

EcSbSubscriptionServer

EcSbSubscription

EcSbNotification

GlStatus

EcSbSubscriptionHandler

4.5.3.8.2.6 Pre-conditions

The subscription server is running normally.

4.5.3.8.2.7 Post-conditions

The subscription server returns to its normal idle state. The expired subscription is gone.

4.5.3.8.2.8 Scenario Description

1. The diagram begins with the daily callback timer firing, which calls CheckExpirations on the EcSbSubscriptionServer object.
2. The EcSbSubscriptionServer searches the list of EcSbSubscription objects for any that have expired, and when it finds one it calls NotifyCancel on that object.
3. This generates a notification that is sent to the user (either direct or email) informing him of the expiration of the subscription.
4. Finally, the subscription is canceled and removed from the server.

4.5.3.8.2.9 Event Trace

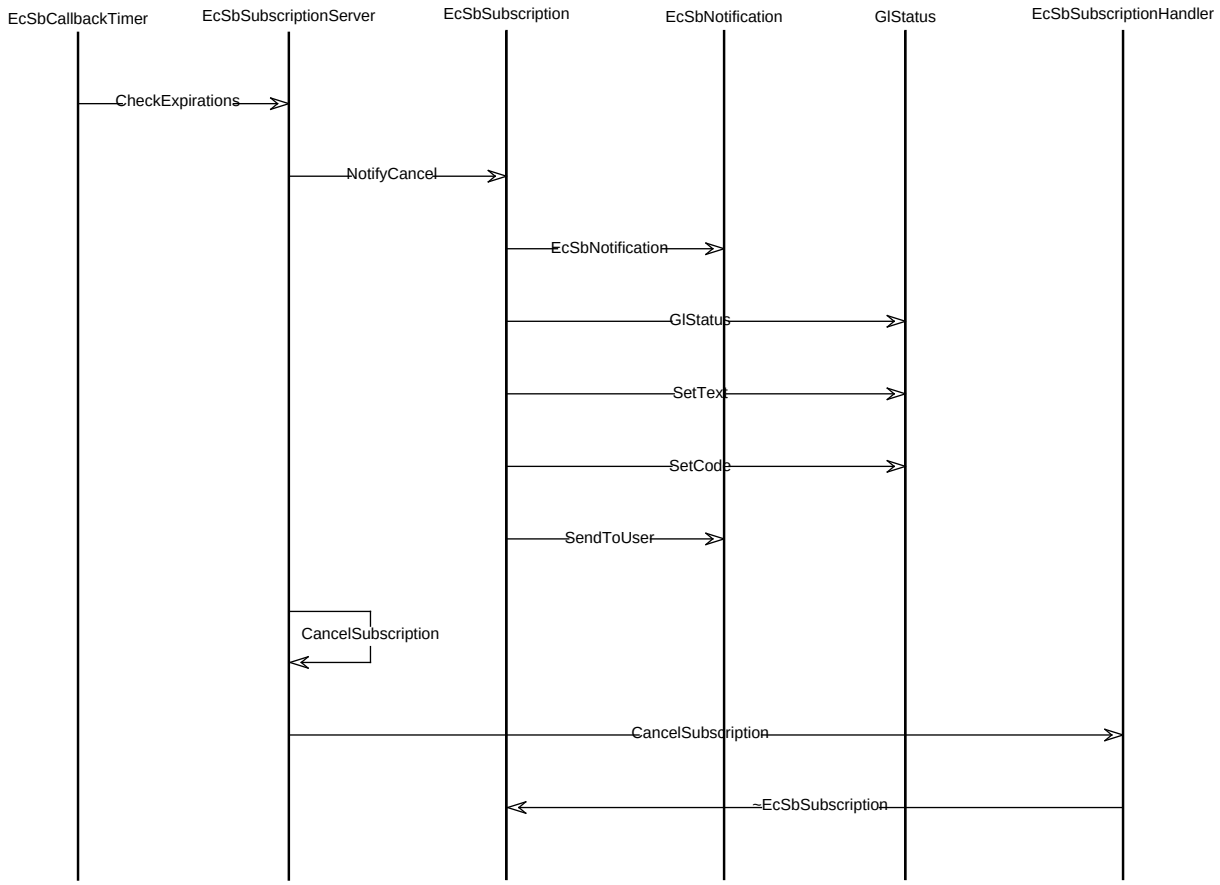

Figure 4.5.3.8-2. Event Trace for Auto Canceling a Subscription

4.5.3.8.3 Scenario #3

4.5.3.8.3.1 Abstract

The scenario shows how the client cancels a subscription.

4.5.3.8.3.2 Interfaces

Client Application using the EcClSubscription object.

4.5.3.8.3.3 Stimulus

The client wishes to withdraw a subscription.

4.5.3.8.3.4 Desired Response

The subscription is canceled and appropriate changes are reflected on the server side.

4.5.3.8.3.5 Participating Classes

EcClSubscription

EcClSubscriptionCollector

EcSbSubscriptionServer

EcSbSubscriptionHandler

EcSbSubscription

4.5.3.8.3.6 Pre-conditions

1. User has logged on and has proper permissions.
2. EcClSubscriptionCollector has already been created.

4.5.3.8.3.7 Post-conditions

The client and server resume normal activity and the subscription is removed from the server's database.

4.5.3.8.3.8 Scenario Description

1. The client issues the Withdraw request. This results in the CancelSubscription method being called on the collector object.
2. The collector then calls CancelSubscription on the server-side EcSbSubscriptionServer.
3. The EcSbSubscriptionServer then calls the CancelSubscription method on the EcSbSubscriptionHandler.

4.5.3.8.3.9 Event Trace

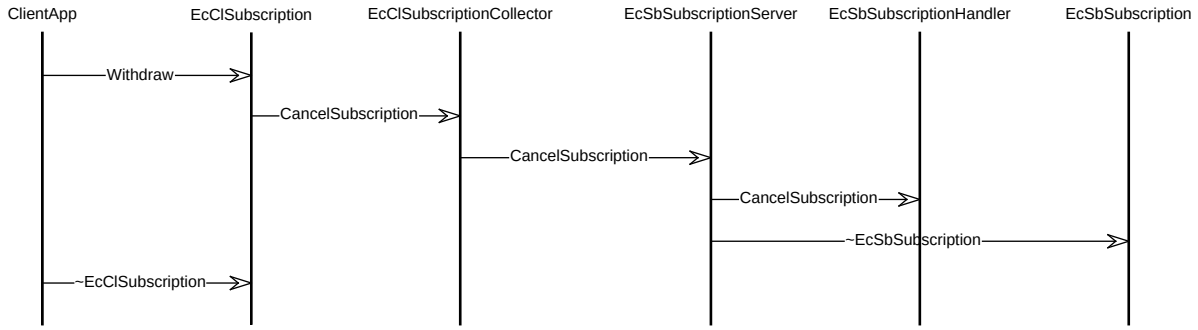

Figure 4.5.3.8-3 Event Trace for Canceling a Subscription

4.5.3.8.4 Scenario #4

4.5.3.8.4.1 Abstract

This scenario shows how a server fulfills a previously submitted subscription. The action for the submitted subscription is to send a notification. The subscription example illustrated here is a one-time subscription. Thus, after the event detection is processed, the subscription will be removed from the active subscription list.

4.5.3.8.4.2 Interfaces

Client applications. In this case relevant datatypes from the Science Data Server.

4.5.3.8.4.3 Stimulus

In this scenario the originating object has detected the advertised and subscribed event.

4.5.3.8.4.4 Desired Response

The event notification is triggered and the subscription is cancelled.

4.5.3.8.4.5 Participating Classes

EcGeESDT

EcGeEventTable

EcSbEvent

EcSbEventHandler

EcSbSubscriptionServer

EcSbSubscriptionHandler

EcSbSubscription

EcSbAction

EcSbNotification

GlStatus

GlParameterList

4.5.3.8.4.6 Pre-conditions

A server is processing a new data item which is a subscribable event.

4.5.3.8.4.7 Post-conditions

The user is notifed and the event is cancelled.

4.5.3.8.4.8 Scenario Description

1. A DSS server looks up the event in the EventTable using Find.
2. The server then triggers the event by calling EcSbEvent.Trigger(). This in turn causes the EventOccured method to be called on the EventHandler.
3. The event handler makes a remote call to the EcSbSubscriptionHandler (ExecuteSubscriptions) to inform the event server that the event has occured.
4. The subscription handler then executes the subscription.
5. The EcSbSubscription object determines the receivers (via GetNotify) and the notification text (via GetText).
6. The notification is then sent via the GINotification object.
7. After sending the notification, the subscription is cancelled via CancelSubscription().

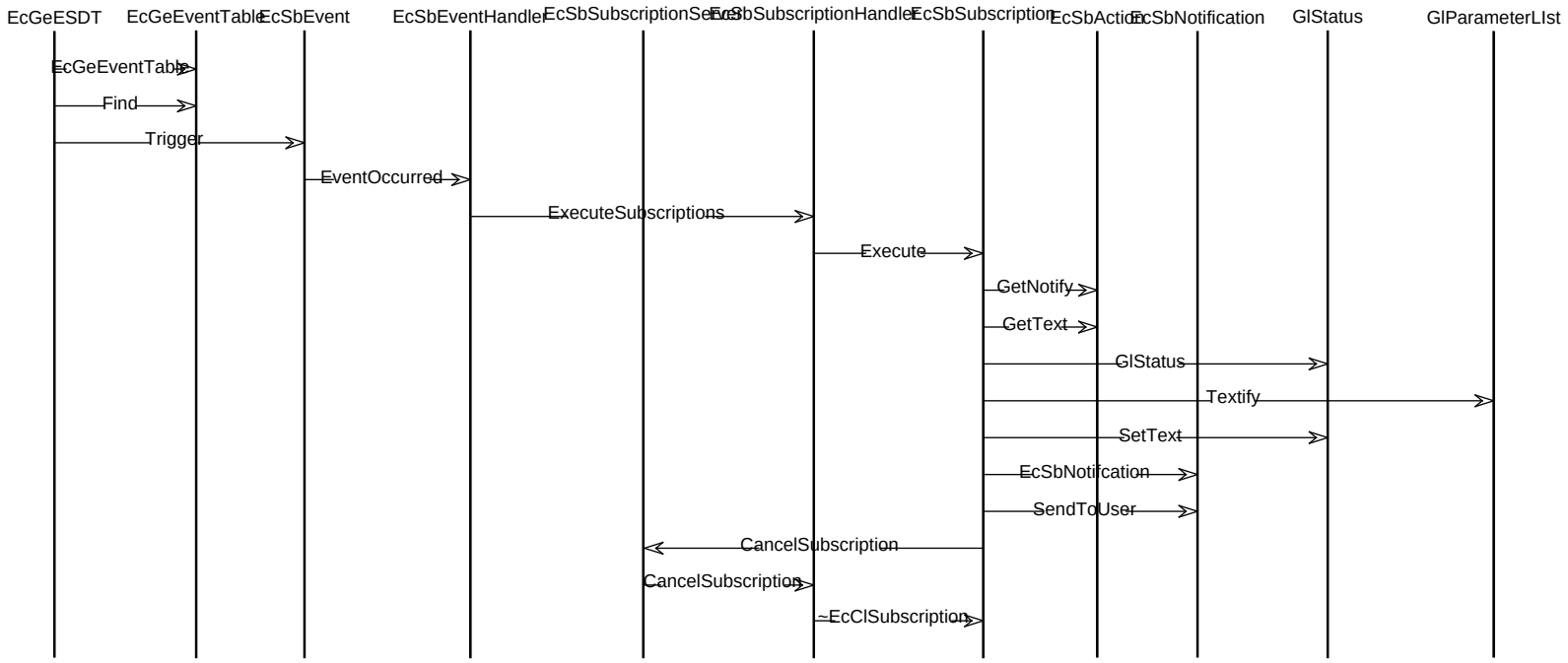

Figure 4.5.3.8-4. Event Trace for Fulfilling a One-Time Subscription

4.5.3.8.5 Scenario #5

4.5.3.8.5.1 Abstract

This scenario shows how a server fulfills a previously submitted subscription. The submitted subscription is for a simple notification to be sent. The subscription example illustrated here is a open-ended subscription, meaning it will continue to be active, waiting for future occurrences of the event.

4.5.3.8.5.2 Interfaces

Client applications. In this case relevant datatypes from the Science Data Server.

4.5.3.8.5.3 Stimulus

In this scenario the originating object has detected the advertised and subscribed event.

4.5.3.8.5.4 Desired Response

The originating object then triggers that event for processing within the Subscription Server.

4.5.3.8.5.5 Participating Classes

EcGeESDT

EcGeEventTable

EcSbEvent

EcSbEventHandler

EcSbSubscriptionHandler

EcSbSubscription

EcSbAction

EcSbNotification

GlStatus

GlParameterList

4.5.3.8.5.6 Pre-conditions

A server is processing a new data item which is a subscribable event.

4.5.3.8.5.7 Post-conditions

The user is notifed and the event is cancelled.

4.5.3.8.5.8 Scenario Description

1. A DSS server looks up the event in the EventTable using Find.
2. The server then triggers the event by calling EcSbEvent.Trigger(). This in turn causes the EventOccured method to be called on the EventHandler.
3. The event handler makes a remote call to the EcSbSubscriptionHandler (ExecuteSubscriptions) to inform the event server that the event has occured.

4. The subscription handler then executes the subscription.
5. The EcSbSubscription object determines the receivers (via GetNotify) and the notification text (via GetText).
6. The notification is then sent via the GINotification object.

4.5.3.8.5.9 Event Trace

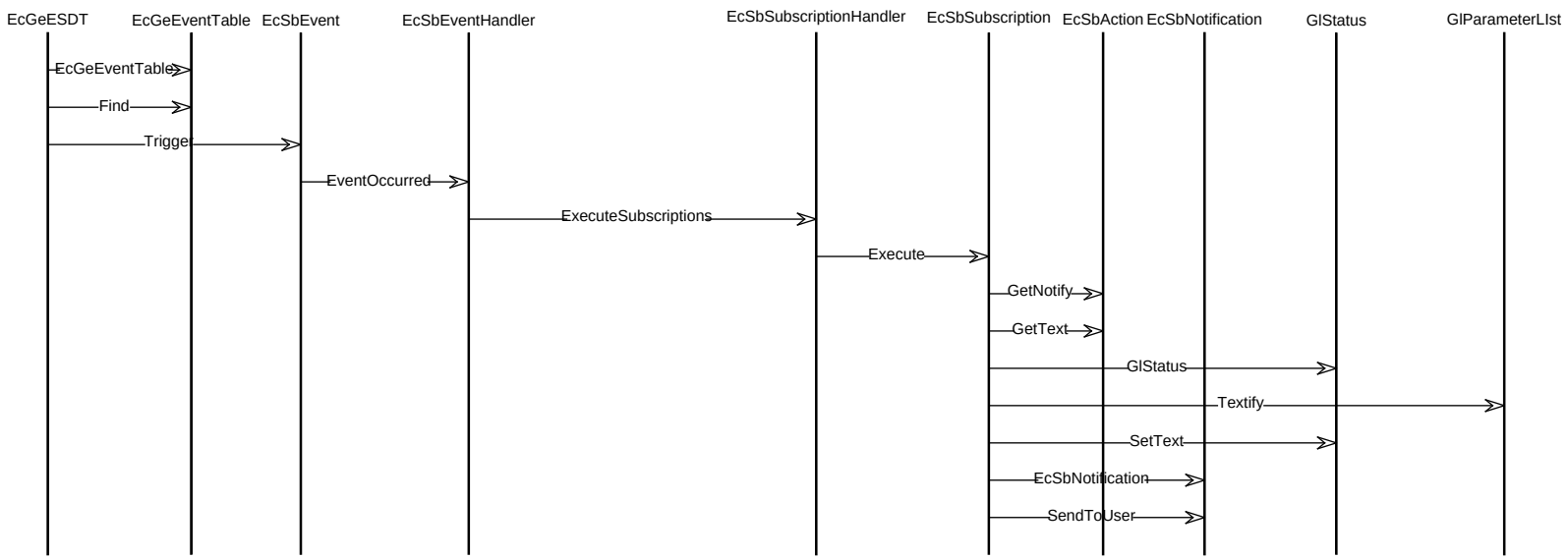

Figure 4.5.3.8-5. Event Trace for Fulfilling An Open-Ended Subscription

4.5.3.8.6 Scenario #6

4.5.3.8.6.1 Abstract

The purpose of this scenario is to show how a subscribable event gets established. This occurs within the context of a variety of scenarios including adding a new Data Type and Startup of the Science Data Server. Although this scenario focuses on events that are generated by Data Types, this is not meant to imply that all subscribable events are Data Type related. A subscribable event might be the occurrence of a disk being full. This scenario begins after an EcDeESDTDescriptor has been created and it has determined that it must establish its subscribable events.

4.5.3.8.6.2 Interfaces

Client applications. In this case relevant datatypes from the Science Data Server and the Advertisement Server.

4.5.3.8.6.3 Stimulus

The Data Server decides that a particular data related event should be subscribable.

4.5.3.8.6.4 Desired Response

The subscribable event is registered with the subscription server.

4.5.3.8.6.5 Participating Classes

EcDeESDTDescriptor

EcDeEventVector

EcDeEvent

EcDbAccess

EcGeESDTEventTable

EcSbFactory

EcSbEvent

EcSbSubscriptionServer

DmAdAdvertisement

4.5.3.8.6.6 Pre-conditions

- The Science Data Sever has the UR of its Subscription Server Factory.
- The Subscription Server is active.
- There is just one event being registered.
- The EcDeEventVector and EcDeEvent are already created.

4.5.3.8.6.7 Post-conditions

The event is registered with the subscription server and an advertisment of the subscribable event is registered with the advertiser.

4.5.3.8.6.8 Scenario Description

1. The EcDeESDTDescriptor establishes a EcGeESDTEventTable for holding all of the events for this Data Type and persistently stores it.
2. Then for an event that the Data Type has the ability to notice, the EcDeESDTDescriptor creates a EcSbEvent with a name, category, and description. Since the EcSbEvent is a distributed object, this causes a creation of an EcSbEvent within the Subscription Server process space.
3. Then the EcDeESDTDescriptor registers the event which notifies the Subscription Server that this is now a known event.
4. The Subscription Server coordinates with the Data Management Subsystem to advertise the fact that this event is subscribable.
5. After the event has been registered, it gets stored persistently in the previously created EcGeESDTEventTable. This process is repeated for each event that this Data Type has the ability to notice.

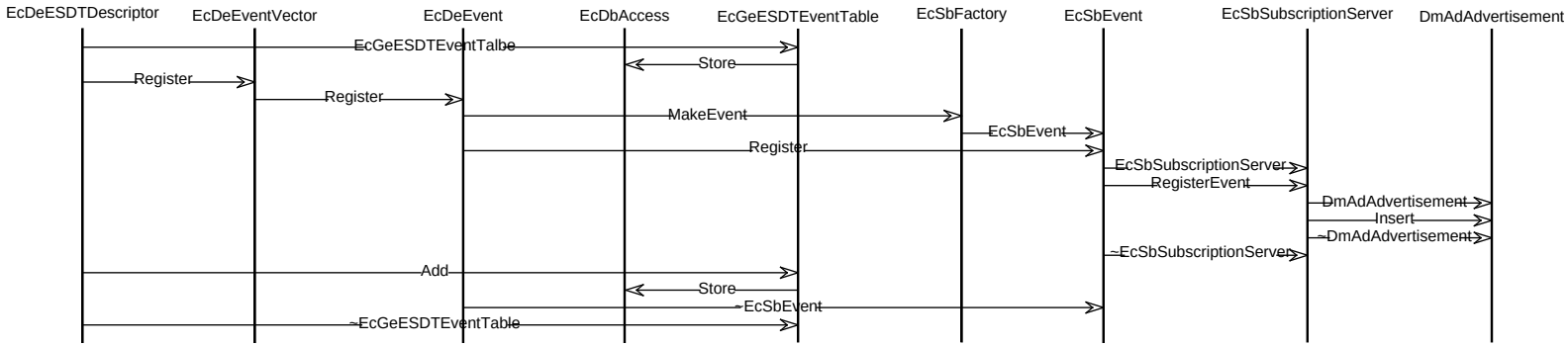

Figure 4.5.3.8-6. Event Trace for Registering a Subscribable Event

4.5.3.8.7 Scenario #7

4.5.3.8.7.1 Abstract

The purpose of this scenario is to show how a subscribable event which has been previously established, gets unregistered. This would be done if a Data Type were being removed from the Data Server. This scenario begins after an EcDeESDTDescriptor has been created and it has determined that it must unregister its subscribable events.

4.5.3.8.7.2 Interfaces

Client applications. Science Data Server and the Advertisement Server in this case.

4.5.3.8.7.3 Stimulus

The Data Server determines that it will not longer support a certain event.

4.5.3.8.7.4 Desired Response

The Data Server acts to remove the event from the Subscription Server.

4.5.3.8.7.5 Participating Classes

EcDeESDTDescriptor

EcGeESDTEventTable

EcDbAccess

EcSbFactory

EcSbEvent

EcSbSubscriptionServer

DmAdAdvertisement

4.5.3.8.7.6 Pre-conditions

- The SDSRV has the UR of its Subscription Server Factory.
- The Subscription Server is active.
- All events of a Data Type are being unregistered.

4.5.3.8.7.7 Post-conditions

The subscribable event is removed from the subscription server and from the advertisements.

4.5.3.8.7.8 Scenario Description

1. The EcDeESDTDescriptor establishes the EcGeESDTEventTable for this Data Type which contains the EcSbEvents that were previously stored when this Data Type was initialized. Because this is a distributed object, an EcSbEvent is created in the Subscription Server execution space.
2. Then for each event that the Data Type has previously stored in the EcGeESDTEventTable, the EcDeESDTDescriptor unregisters that event. This causes the SubscriptionServer to cancel the advertisement associated with this event.
3. Finally, the EcSbEvent is removed from the persistent EcGeESDTEventTable.

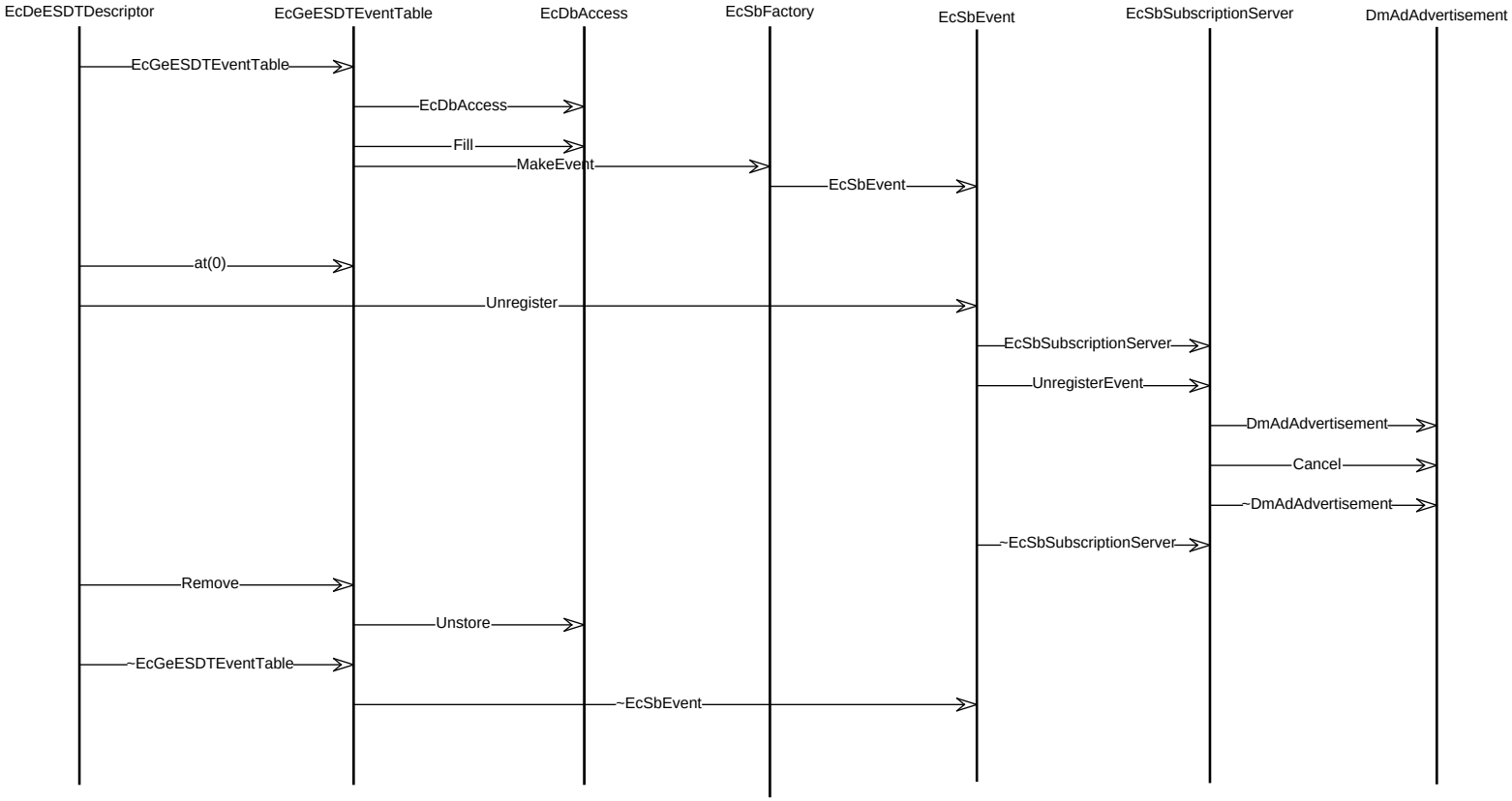

Figure 4.5.3.8-7. Event Trace for Unregistering a Subscribable Event

4.5.3.8.8 Scenario #8

4.5.3.8.8.1 Abstract

The scenario shows how the client updates a subscription.

4.5.3.8.8.2 Interfaces

Client Applications.

4.5.3.8.8.3 Stimulus

The client wishes to update a subscription.

4.5.3.8.8.4 Desired Response

The subscription is updated.

4.5.3.8.8.5 Participating Classes

EcClSubscription

EcClSubscriptionCollector

Other classes included in scenario #1 and scenario #4 are also needed here to complete the scenario.

4.5.3.8.8.6 Pre-conditions

- user has logged on and has proper permissions.
- EcClSubscriptionCollector has already been created.

4.5.3.8.8.7 Post-conditions

The subscription is still registered, but it has been updated.

4.5.3.8.8.8 Scenario Description

The scenario shows steps used to update a subscription from the client side and ultimately, reflected on the server side. This is achieved through the distributed object, EcClSubscriptionCollector. The object, EcClSubscriptionCollector, has already been constructed. This scenario shows that the client wants to update the expiration date of the subscription. These are the steps:

1. The client sets the expiration date of the subscription.
2. The older subscription is withdrawn as outlined in the scenario #4, subscription server canceling a subscription.
3. The subscription is submitted as outlined in the scenario #1, subscription server submitting a subscription.

4.5.3.8.8.9 Event Trace

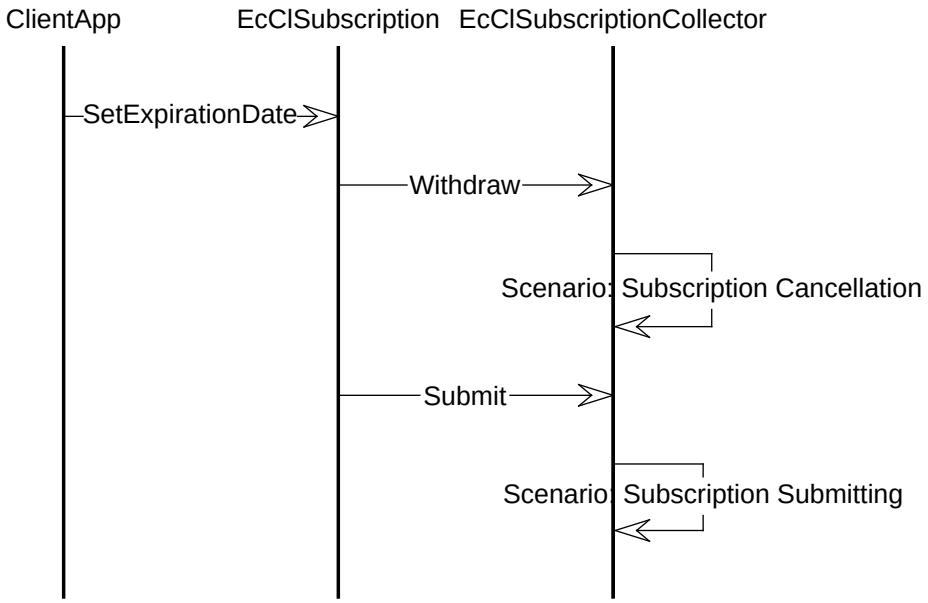

Figure 4.5.3.8-8. Event Trace for Updating a Subscription

4.5.3.8.9 Scenario #9

4.5.3.8.9.1 Abstract

This scenario shows how the EcClSubscriptionCollector is populated with the subscriptions for a specified user.

4.5.3.8.9.2 Interfaces

Client applications. Science Data Server is used as an example in this case.

4.5.3.8.9.3 Stimulus

The client may wish to get a list of subscriptions.

4.5.3.8.9.4 Desired Response

The list is returned.

4.5.3.8.9.5 Participating Classes

EcClSubscriptionCollector

EcClSubscription

EcSbSubscriptionServer

GlParameterList

GlStringP

4.5.3.8.9.6 Pre-conditions

There is one subscription for the user.

4.5.3.8.9.7 Post-conditions

All steps execute successfully.

4.5.3.8.9.8 Scenario Description

The client tells the EcClSubscriptionCollector (a distributed object) to BuildList, which (server-side) asks the EcSbSubscriptionServer to build a parameter list of references to the subscriptions for a user. The client-side EcClSubscriptionCollector takes this list and builds a EcClSubscription (only one in this case) from it and adds it to himself.

4.5.3.8.9.9 Event Trace

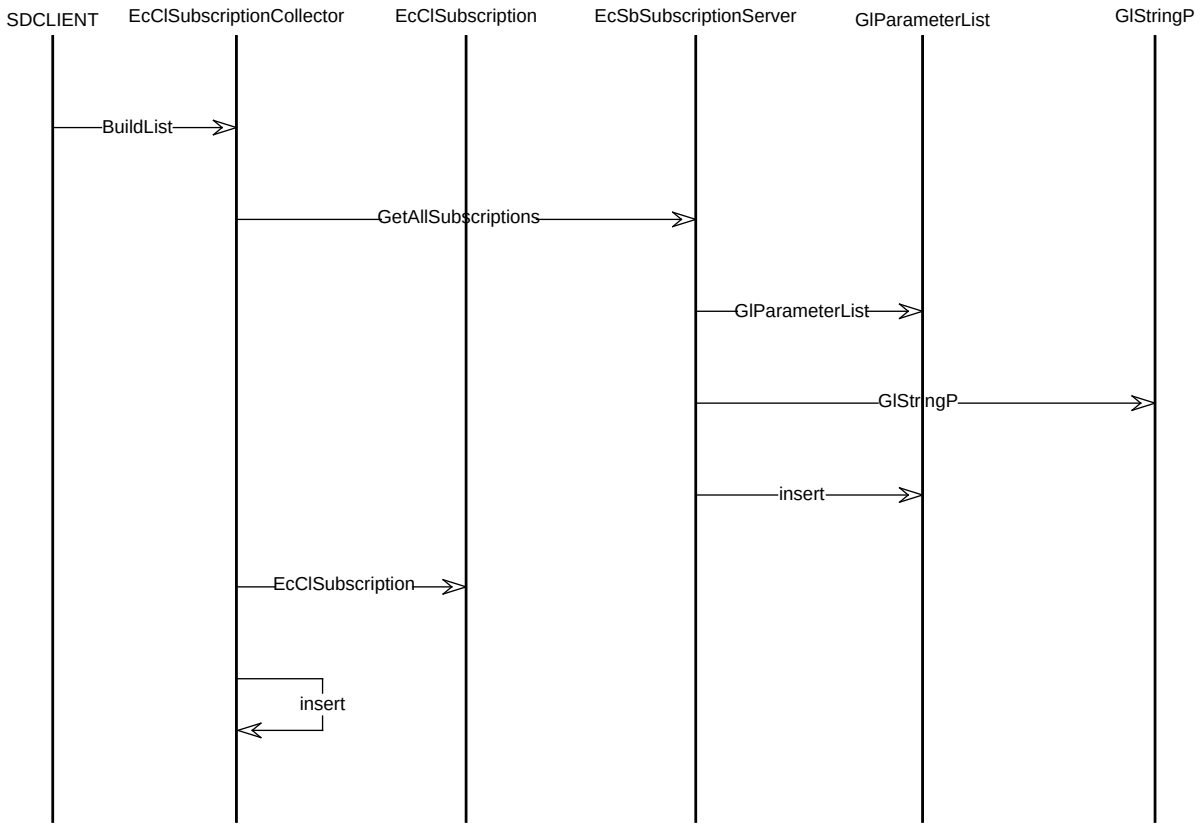

Figure 4.5.3.8-9. Event Trace for Returning a List of Subscriptions

4.5.3.9 Implementation

The Subscription Server depends on the implementations of various ADSRV, CSS and MSS services.

4.5.3.10 Service/CSCI Management and Operation

4.5.3.10.1 System Management and Strategy

The SubscriptionServer Process comprises the following CSS (DCCI) CSCs:

- Client
- Configuration/Startup
- Global
- Subscription

It is also comprised of the interface class CSCs from ADSRV and MSS.

4.5.3.10.2 Operator Interface

Not applicable.

4.5.3.10.3 Reports

None.

4.5.4 Universal References

4.5.4.1 Overview

Universal References (URs) provide applications and users a system wide mechanism for referencing ECS data and service objects. Once a UR is made for an object, the object can be disposed of and later reconstituted from the UR. URs can refer to objects that may be local to an address space, or remote.

URs are implemented as a framework which provides objects the capability to create URs for themselvers, distribute URs throughout the system, then use these URs to reconstitute and/or access the original object. URs themselves are objects which can externalize themselves into an ASCII representation, then internalize from their ASCII representation back to objects. The content of the externalized UR is human transcribable and transport friendly. While the UR mechanism guarantees reliable data externalization and internalization, the content of each type of UR is application specific. Only the object that initially provides the UR (from now on we will call this the "UR Provider") is allowed to access and understand its content. URs are strongly typed to enforce appropriate access control to internal data both at compile time and during runtime. Since URs are typed and have object specific data in them, separate UR object classes exist for each UR Provider object class referred to. All of these UR classes use the mechanisms provided by the UR framework. The framework also provides UR Providers support for their common requirements. The following diagram, Figure 4.5.4.1-1, illustrates the functions of the various frameworks.

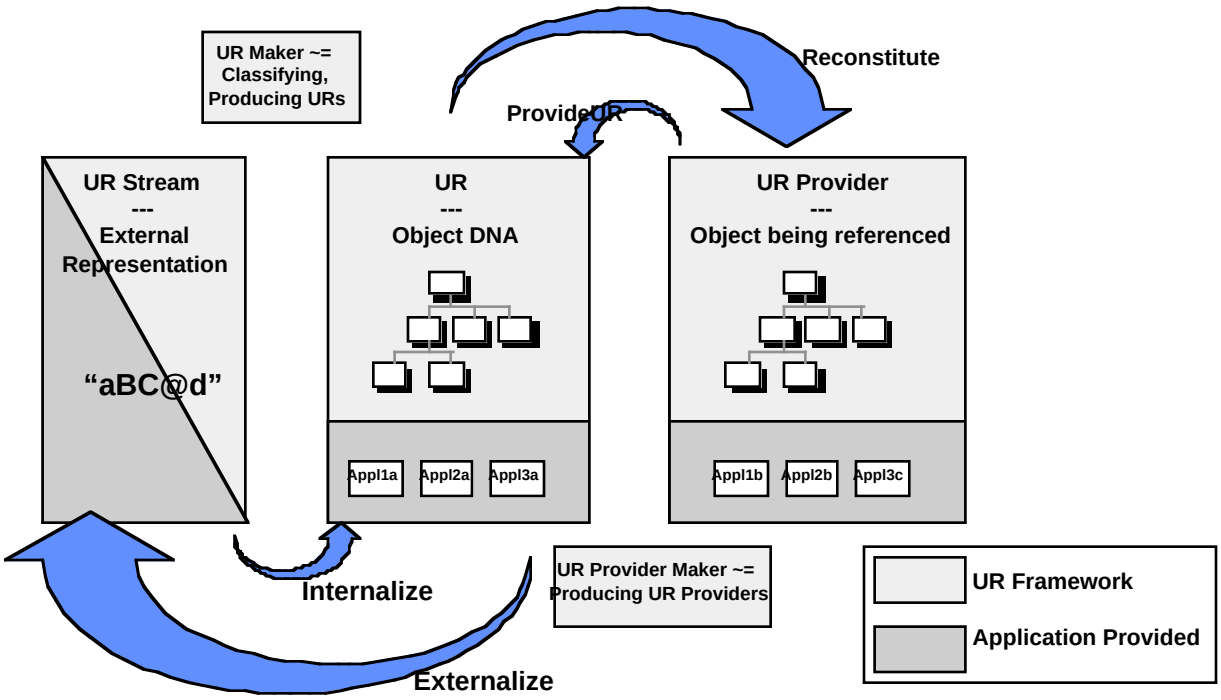

Figure 4.5.4.1-1. Functions of Different UR Frameworks

From left to right, going from top to bottom,

- the UR Provider framework supports common functions for all object referenced by URs. This framework allows URs for the UR Provider to be extracted and later reconstituted.
- ECS application objects subclass this framework to enable themselves to be referred to by URs.
- the UR framework supports common functions for referring to UR Providers; this function will allow URs to be externalized and internalized.
- ECS application UR objects subclass this framework and will hold the application specific state data to uniquely identify the application object referred to.
- URs can be externalized to ASCII strings that are managed by the user.

The object paradigm uses techniques of abstraction and inheritance to create designs that are resilient to change. ECS will extend the data and services it provides. The UR and UR Provider Frameworks are designed to support both abstraction and inheritance. An abstract UR that provides a defined set of data and services can be passed to a UR consumer. The consumer is not required to know the actual concrete UR object they are working with. When these abstract URs are used to access the object they refer to, the concrete object is accessed. However, the consumer is not required to know which concrete object they are using. They are only required to know the interfaces to the abstract object. This capability of working with abstract URs and abstract objects allows ECS to extend, through inheritance, the data and services it provides with minimal changes

to applications. The framework supports the dynamic introduction of new remote objects that meet a given pre-existing abstract interface.

A corollary design component of URs concerns backward compatibility. Objects providing URs will change over time. The URs they provide as references to themselves will then also probably change. The UR and UR Provider Frameworks allow a UR Provider to support multiple UR types as long as a backward compatibility feature exists in the provider. This design component will mitigate problems of end users and applications using URs dispensed from prior software versions.

As described, URs have an extensible and evolvable design which allows the UR types to grow and change over time. A very important specialization of URs is used in the SRF design, where URs are tailored to support location of services. The UR framework is independent of the communication infrastructure and does not couple the ECS architecture to OODCE. It provides a general mechanism for keeping references to logical entities for long periods of time, without the computational costs of maintaining those objects in memory.

4.5.4.2 Context

ECS functions are performed by manipulating logical entities represented at runtime as C++ objects in virtual memory. Users and applications require references to the logical entities beyond the time that it is computationally effective to keep the objects in memory. Therefore, applications and users are given Universal References (URs) to these objects. URs have a low cost to keep in memory and can be externalized into an ASCII string that an end user can manage. A UR has the capability of re-accessing and/or reconstituting the object into memory as needed. Therefore, the object does not have to remain in memory, and can if appropriate, be written to a secondary storage system, like a database.

4.5.4.3 Object Model

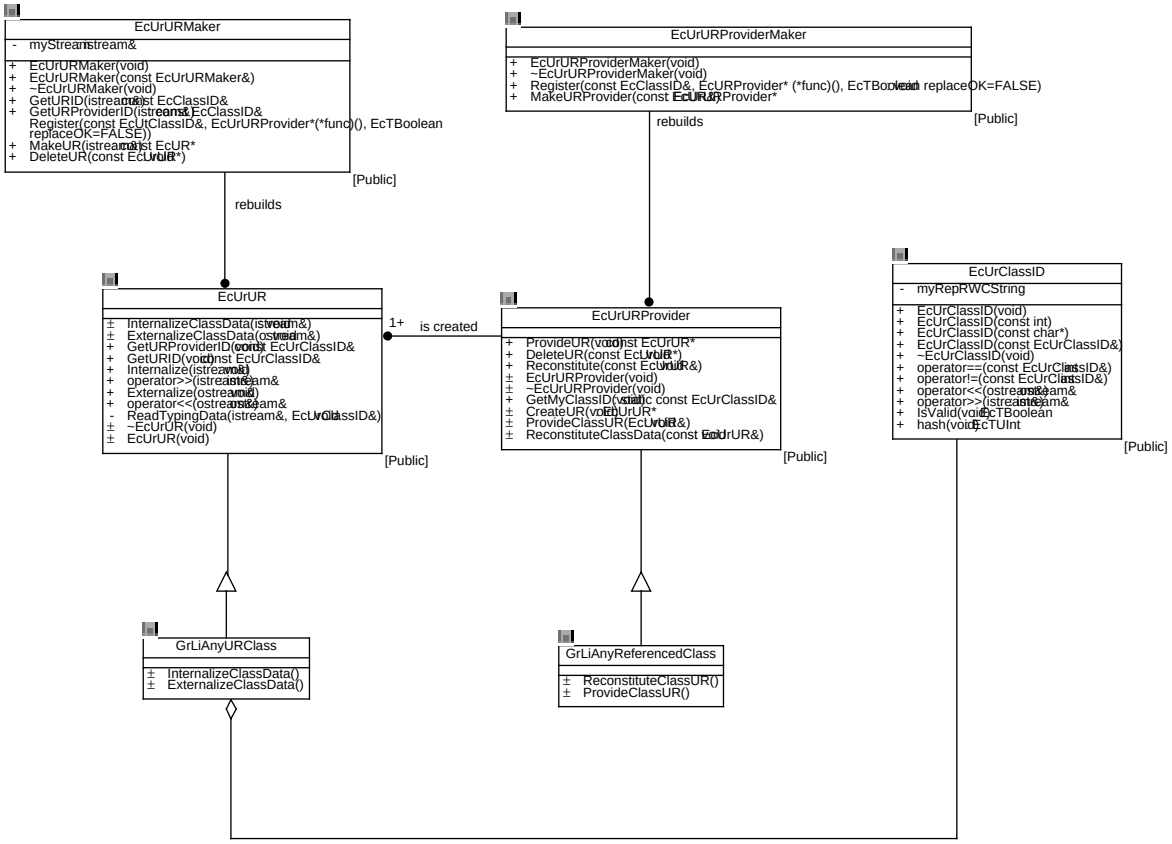

Figure 4.5.4.3-1. Universal References Object Model

4.5.4.3.1 EcUrClassID Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class encapsulates an indentifier for ECS C++ classes. It supports several constructors, a mechanism for comparing two instance of this class, and the ability to read/ write itself from streams. A const global object of this class, "theInvalidClassID" is defined. It can be used to set/check if something is out of range.

Attributes:

myRep - This attribute is the internal representation of the Class ID.
Data Type:RWCString
Privilege:Private

Default Value:

Operations:

EcUrClassID - This method is the default constructor for this class. This object's value, while initialized, is still undefined.
Arguments:void
Return Type:Void
Privilege:Public

EcUrClassID - This method will construct an object whose value is defined by the argument.
Arguments:const int
Return Type:Void
Privilege:Public

EcUrClassID - This method will construct an object whose value is defined by the argument.
Arguments:const char*
Return Type:Void
Privilege:Public

EcUrClassID - This method is the copy constructor the object.
Arguments:const EcUrClassID&
Return Type:Void
Privilege:Public

IsValid -
Arguments:void
Return Type:EcTBoolean
Privilege:Public

hash -
Arguments:void
Return Type:EcTUInt
Privilege:Public

operator!= - This method is the inequality operator. It will return a non-zero value if the objects being compared are logically different.
Arguments:const EcUrClassID&
Return Type:int
Privilege:Public

operator<<

Arguments:ostream&
Return Type:ostream&
Privilege:Public

operator== -
Arguments:const EcUrClassID&
Return Type:int
Privilege:Public

operator>> - This is the stream extraction operator for the method. Unfortunately, while the name 'extraction' is similar to the UR operation 'Extract' its behavior is unrelated. This method reads the contents of the class ID from the input stream argument. It can only read a class ID written with the insertion operator<<. An exception will be raised if the stream or stream data is invalid.
Arguments:istream&
Return Type:istream&
Privilege:Public

~EcUrClassID - This method is the destructor for the object. State cleanup before the object is destroyed will occur here.
Arguments:void
Return Type:Void
Privilege:Public

Associations:

The EcUrClassID class has associations with the following classes:
GrLiAnyURClass (Aggregation)

4.5.4.3.2 EcUrUR Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This is the abstract base class for all Universal Reference (UR)s. A UR is a special ECS identifier for an object. What makes it special is that an object can be identified, but the object does not have to exist in memory at the time. The contents of a UR are specified by subclasses. Generally speaking, the contents are the key elements of the object that this UR refers to. It can be thought of as DNA. We can reconstitute or clone an organism (i.e. object or URProvider) given its DNA (i.e. UR). The key public methods are "Externalize" and "Internalize"

Attributes:

None

Operations:

EcUrUR - This protected constructor initializes the object.
Arguments:void
Return Type:Void
Privilege:Protected

Externalize - This method is the public interface for exporting the contents of the UR to an output stream. This data can be imported later with Internalize. This method allows URs to persist outside the runtime of the application. The stream will contain information (processed by Internalize) that prevents tampering.
Arguments:ostream&
Return Type:void
Privilege:Public

ExternalizeClassData - This method exports the UR's state to the output stream argument. This method shall be overridden by all concrete UR classes. The data written should have been read back from the stream with InternalizeClassData. This method shall call all appropriate associated objects' ExternalizeClassData method. An associated object is either a direct base class or a contained object. Note the public interface for this functionality is Externalize.
Arguments:ostream&
Return Type:void
Privilege:Protected

GetURID - This method returns the class ID of the UR object. Often abstract base class URs are passed around. This method can be used to find the actual conrete object being passed. The method GetURProviderID can be used to findout what concrete object this UR refers to.
Arguments:void
Return Type:const EcUrClassID&
Privilege:Public

GetURProviderID - The method returns the Class ID of the object that is referred to by this UR. This is the object that Extract'ed this UR and that can Reconstitute itself from this UR.
Arguments:void
Return Type:const EcUrClassID&
Privilege:Public

Internalize - This is the public method for importing data from a stream into a UR. If the stream does not contain the correct kind of data for this UR or if the data is invalid, an

exception will be raised.
Arguments:istream&
Return Type:void
Privilege:Public

InternalizeClassData - This method imports the UR's state from the input stream argument. This method shall be overridden by all concrete UR classes. The data read should have been written to the stream with ExternalizeClassData. This method shall call all appropriate associated objects InternalizeClassData method. An associated object is either a direct base class or a contained object. Note the public interface for this functionality is Internalize. Exceptions should be raised in this operation if errors occur.
Arguments:istream&
Return Type:void
Privilege:Protected

ReadTypingData - This private method allows a friend object class to determine the UR class ID contained in a stream. It is used by URMaker. The class ID is placed in the Class ID reference argument.
Arguments:istream&, EcUrClassID&
Return Type:void
Privilege:Private

operator<< - This method the same as the Externalize method except it support the standard stream insertion signature.
Arguments:ostream&
Return Type:ostream&
Privilege:Public

operator>> - This method the same as the Internalize method except it support the standard stream extraction signature.
Arguments:istream&
Return Type:istream&
Privilege:Public

~EcUrUR - This protected method is the destructor for the object. It will clean up state prior to the object being destroyed.
Arguments:void
Return Type:Void
Privilege:Protected

Associations:

The EcUrUR class has associations with the following classes:

Class: EcUrURMaker
Class: EcUrURProvider

4.5.4.3.3 EcUrURMaker Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class supports two correlated responsibilities. First, it is an object factory for Universal Reference (UR)s. It allows subclasses of URs to register themselves. Then based on a given encapsulated ClassID, it can dynamically construct URs of any registered type. Secondly, it can decode a stream containing externalized (i.e. ASCII represented) URs. This class can read a stream containing a UR and identify the UR specified in the stream or the UR Provider referred to by the UR in the stream.

Attributes:

myStream - This attribute is a pointer to the current input stream associated with this object. This stream contains exported URs that this object helps to import.
Data Type:istream&
Privilege:Private
Default Value:

Operations:

DeleteUR - This static function is provided as an aid to callers who used MakeUR to create a "const EcUR*. While they are responsible for deleting it, they can't because it is const. This routine can delete it.
Arguments:const EcUrUR*
Return Type:void
Privilege:Public

EcUrURMaker - This method is the default constructor for the object. The stream still must be set after this constructor.
Arguments:void
Return Type:Void
Privilege:Public

EcUrURMaker
Arguments:const EcUrURMaker&
Return Type:Void
Privilege:Public

GetURID - This method returns a reference to the Class ID of the exported UR currently at the beginning of the input stream. The returned class ID will be invalid if the stream does not contain a valid UR.
Arguments:istream&
Return Type:const EcClassID&
Privilege:Public

GetURProviderID - This method returns a reference to the Class ID of the UR Provider object referred to by the exported UR currently at the beginning of the input stream. An invalid class ID will be returned if the stream does not contain a valid UR.
Arguments:istream&
Return Type:const EcClassID&
Privilege:Public

MakeUR - Makes a UR and Internalize it. The user is responsible for deleting the return value through ::DeletedUR. This function can return NULL if the UR defined in the stream is not registered.
Arguments:istream&
Return Type:const EcUR*
Privilege:Public

Register
Arguments:const EcUtClassID&, EcUrURProvider*(*func)(), EcTBoolean replaceOK=FALSE

~EcUrURMaker - This method is the destructor for the object. It will clean up the state to allow proper deallocation. The state of the internal stream will not be affected by this operation.
Arguments:void
Return Type:Void
Privilege:Public

Associations:

The EcUrURMaker class has associations with the following classes:
Class: EcUrUR

4.5.4.3.4 EcUrURProvider Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class is the abstract base class for all things refered to by Universal Reference (UR)s.

Its primary responsibility is to provide URs to clients, thus the name "UR Provider". The primary operations of interest are "ProvideUR" and "Reconstitute".

Attributes:

None

Operations:

CreateUR - This method will create and return an new UR that can refer to this object. This method shall be overridden by all concrete derived classes of this class.
Arguments:void
Return Type:EcUrUR*
Privilege:Protected

DeleteUR - This function is static. Since we return a const UR, from ProvideUR, the client can't delet it. This method is responsible for deleting it.
Arguments:const EcUrUR*
Return Type:void
Privilege:Public

EcUrURProvider - This method is the constructor for the class.
Arguments:void
Return Type:Void
Privilege:Protected

GetMyClassID -
Arguments:void
Return Type:static const EcUrClassID&
Privilege:Public

ProvideClassUR - Provide primary key data for the class state and place it in the UR. Then, call this method for each of your associated objects.
Arguments:EcUrUR&
Return Type:void
Privilege:Protected

ProvideUR - This method will provide a Universal Reference to the caller that represents the current logical entity. The return value is allocated on the heap and should be deallocated with the "DeleteUR" method. The return value is a "Memento" (standard design pattern) that can be used to logically bring this object back. The "Reconstitute" method can be called to bring an object back to this state. Note that state is application specific. Derived classes should decide on policies. Possibilities include bring back that exact object the UR came from or reconstituting to the latest version. This method is a

"template method" (standard design pattern) that calls "ProvideClassUR".
Arguments:void
Return Type:const EcUrUR*
Privilege:Public

Reconstitute - Public method to make ourselves the object that is logically referred to by the UR.
Arguments:const EcUrUR&
Return Type:void
Privilege:Public

ReconstituteClassData - Reconstitute class data self based on the UR. Then, call this for associated objects.
Arguments:const EcUrUR&
Return Type:void
Privilege:Protected

~EcUrURProvider - This method is the destructor for the class.
Arguments:void
Return Type:Void
Privilege:Protected

Associations:

The EcUrURProvider class has associations with the following classes:
Class: EcUrUR
Class: EcUrURProviderMaker

4.5.4.3.5 EcUrURProviderMaker Class

Parent Class:Not Applicable
Public:Yes
Distributed Object:No
Purpose and Description:
This class is an object factory responsible for the registration and dynamic creation of object subclasses from "URProvider". Objects are indexed by the encapsulated type "ClassID".

Attributes:

None

Operations:

EcUrURProviderMaker - This method is the constructor for this object.
Arguments:void
Return Type:Void
Privilege:Public

MakeURProvider - Make a URProvider that matches the UR argument and then Reconstitute it.
Arguments:const EcUR&
Return Type:EcUrURProvider*
Privilege:Public

Register - Register a creation function for a UR Provider derived object.
Arguments:const EcClassID&, EcURProvider* (*func)(), EcTBoolean replaceOK=FALSE
Return Type:void
Privilege:Public

~EcUrURProviderMaker - This method is the destructor for this method.
Arguments:void
Return Type:Void
Privilege:Public

Associations:

The EcUrURProviderMaker class has associations with the following classes:
Class: EcUrURProvider

4.5.4.3.6 GrLiAnyReferencedClass Class

Parent Class:EcUrURProvider
Public:No
Distributed Object:No
Purpose and Description:
This class is an example of an application specific UR provider class.

Attributes:

All Attributes inherited from parent class

Operations:

ProvideClassUR - This method is the application specific override of the parent class pure virtual method.
Arguments:

Return Type:Void
Privilege:Protected

ReconstituteClassUR - This method is the application specific override of the parent class pure virtual method.
Arguments:
Return Type:Void
Privilege:Protected

Associations:

The GrLiAnyReferencedClass class has associations with the following classes:
None

4.5.4.3.7 GrLiAnyURClass Class

Parent Class:EcUrUR
Public:No
Distributed Object:No
Purpose and Description:
This class is an example of an application specific UR class

Attributes:

All Attributes inherited from parent class

Operations:

ExternalizeClassData - This method is the application specific override of the parent class pure virtual method.
Arguments:
Return Type:Void
Privilege:Protected

InternalizeClassData - This method is the application specific override of the parent class pure virtual method.
Arguments:
Return Type:Void
Privilege:Protected

Associations:

The GrLiAnyURClass class has associations with the following classes:

None

4.5.4.4 Functional Model

The Universal Reference Framework provides a set of classes with distinct functionality to facilitate the incorporation of Universal References into ECS client/server applications. The classes can be categorized into three distinct catogories:

(1) UR framework classes which provide a framework for the URs themselves

(2) UR Provider framework classes which provide a framework for providers of URs, and

(3) UR utility classes which support the UR and UR provider framework classes.

UR Framework Classes

The class EcUrUR is the abstract base class for all Universal References. A UR is a special ECS identifier for an object. What makes it special is that an object can be identified, but the object does not have to exist in memory at the time. The contents of a UR are specified by subclasses. Generally speaking, the contents are the key elements of the object to which this UR refers. It can be thought of as DNA. We can reconstitute or clone an EcUrURProvider object given its DNA (i.e. UR).

Developers subclass the EcUrUR to define UR classes which will provide references to data and services specific to their applications. These derived classes are provided with the attributes and operations necessary to reconstitute the UR referrent i.e., the UR provider. An example of this specialization is the class GrLiAnyURClass. As shown in the object model GrLiAnyURClass contains two operations, Externalize and Internalize. These two operations are replacements for the corresponding pure virtual functions in the abstract class, and as such must be developed at the subclass level. Other operations for GrLiAnyURClass are inherited from the parent class.

The key public methods of EcUrUR are Externalize and Internalize. Externalize provides the capability for the UR to convert itself into an ASCII stream. In this form the UR becomes accessable by other types of applications (which could be email, or an ASCII reader). Internalize provides the UR object with the capability to accept an externalized form of the UR and store it within itself.

UR Provider Framework

The class EcUrURProvider is the abstract base class for all things referred to by Universal Reference (UR)s. Its primary responsibility is to provide URs to clients, thus the name "UR Provider".The primary operations of interest are "ProvideUR" and "Reconstitute". The ProvideUR operation returns to the caller a UR object which is the referrent to the UR provider. The Reconstitute operation rebuilds the provider using the UR passed to it.

Developers specialize the EcUrURProvider class to develop subclasses for the object which they want to be able to reference and store using URs. These developer created provider subclasses represent objects which can exist in different address spaces from the URs which refer to them. An example of this specialization is the class GrLiAnyReferencedClass which contains two operations, ProvideClassUr and ReconstituteClassUr. These two operations provide implementation for the corresponding pure virtual functions in the parent class, and as such must

be developed at the subclass level. Other operations for GrLiAnyReferencedClass are inherited from the parent class.

Utility Classes

The EcUrClassID class encapsulates an identifier for ECS C++ classes. It supports several constructors, a mechanism for comparing two instances of this class, and the ability to read/write itself from streams. A const global object of this class, "theInvalidClassID" is defined and may be used to set/check if parameters are out of range.

The class EcUrURMaker supports two correlated responsibilities. First, it is an object factory for Universal Reference (UR)s. It allows subclasses of URs to register themselves. Then based on a given encapsulated ClassID, it can dynamically construct URs of any registered type. Secondly, it can decode a stream containing externalized (i.e. ASCII represented) URs. This class can read a stream containing a UR and identify the UR specified in the stream or the UR Provider referred to by the UR in the stream, so that the referenced object can be rebuilt.

The class EcUrURProviderMaker is an object factory responsible for the registration and dynamic creation of object subclasses from "URProvider". Objects are indexed by the encapsulated type "ClassID"

Table 4.5.4.4-1 describes the UR Framework classes.

Table 4.5.3.4-1. Universal References Object Responsibility Matrix (1 of 2)

Class name	Description
EcUrUR	The abstract base class for all Universal Reference (UR)s. Must be specialized by developers to provide UR classes which are application specific.
EcUrURProvider	The abstract base class for all data and service objects referred to by Universal Reference (UR)s. Must be specialized by developers to provide specific provider classes.
EcUrClassID	This class supports a mechanism for comparing two instances of this class, and the ability to read/write itself from streams.
EcUrURMaker	An object factory for URs. Provides several capabilities: 1. Registering of UR object classes, 2. Creation of URs based on registered object class type, 3. Read streams with URs and rebuild the UR objects.

Table 4.5.3.4-1. Universal References Object Responsibility Matrix (2 of 2)

Class name	Description
EcUrURProviderMaker	An object factory for UR providers. Provides capability to register data and services object classes, and refulds UR provider objects using URs.
GrLiAnyUR	Example of a specialization of the EcUrUR class. Any class referenced by a UR must define its own UR, which is a specialization of the EcUrUR class.
GrLiAnyReferencedClass	Example of a specialization of the EcUrURProvider class. Any class referenced by a UR must have its own specialization of the EcUrURProviders class.

4.5.4.5 Dynamic Model Scenarios

Two main scenarios describe how the UR framework supports the use of URs. The first scenario involves the creation of URs: given an object, how is a UR created for it and then externalized (in a e-mailable form for example). Once a UR is created for an object and externalized, then a second scenario involves the reverse process: going from the external representation of an UR to create the object which the UR describes.

The following event traces illustrate the steps in these two scenarios. The event trace for scenario one illustrates the use of the UR framework by a referenced class to create a UR for itself, and provide an external representation of the UR. The event trace for the second scenario illustrates the use of the UR framework to recreate a object from a externalized UR.

Both scenarios are purposefully generic to focus on the functioning of UR Framework components. More detailed scenarios could be developed for subsystem-specific specializations of the URs, but these scenarios would be derived from either of the two scenarios presented here.

4.5.4.5.1 Scenario #1

4.5.4.5.1.1 Abstract

This scenario shows how a referenced class creates a UR to itself, and provides an external representation of the UR.

4.5.4.5.1.1 Interfaces

Client application shown as a ServiceUser object in the event trace.

4.5.4.5.1.2 Stimulus

An application is required to provide a UR in a transportable format.

4.5.4.5.1.3 Desired Response

An ASCII version of the UR for the referenced object is made available.

4.5.4.5.1.4 Participating Classes

GrLiAnyURClass

GrLiAnyURProviderClass

4.5.4.5.1.5 Pre-conditions

None.

4.5.4.5.1.6 Post-conditions

None.

4.5.4.5.1.7 Scenario Description

The event trace starts with the ServiceUser object making a call to a referenced object (GrLiAnyReferencedClass) to provide a UR for itself. The referenced object is instantiated from a child class of the EcUrURProvider abstract class. To provide a more concrete example from ECS, session classes, data granule classes, and advertisement classes will all be among those specialized from EcUrURProvider. Objects instantiated from any of them could be substituted in the example for aGrLiAnyReferencedClass as providers of URs.

The ServiceUser object invokes the method ProvideUr of the object GrLiAnyReferencedClass when it wants to create the UR for that object. This call causes the creation of the UR object, an object of type GrLiAnyUrClass. In the event trace diagram only the call to the constructor is shown. Operations to set the UR values are specific for every UR and are not shown here. The last step in the event trace is the service provider's call to the GrLiAnyURClass object to externalize itself, using the method Externalize(ostream&). This method will provide back an ASCII representation of UR object, which can then be easily transported or stored in secondary storage.

GrLiAnyURClass is a child class of the EcUrUR base class. Also as with the referenced class, session UR classes, data UR classes, advertisement UR classes will all be among those specialized from EcUrUR. Objects instantiated from any of them could be substituted in the example for GrLiAnyUrClass object as types of URs for the appropriate referenced object.

4.5.4.5.1.8 Event Trace

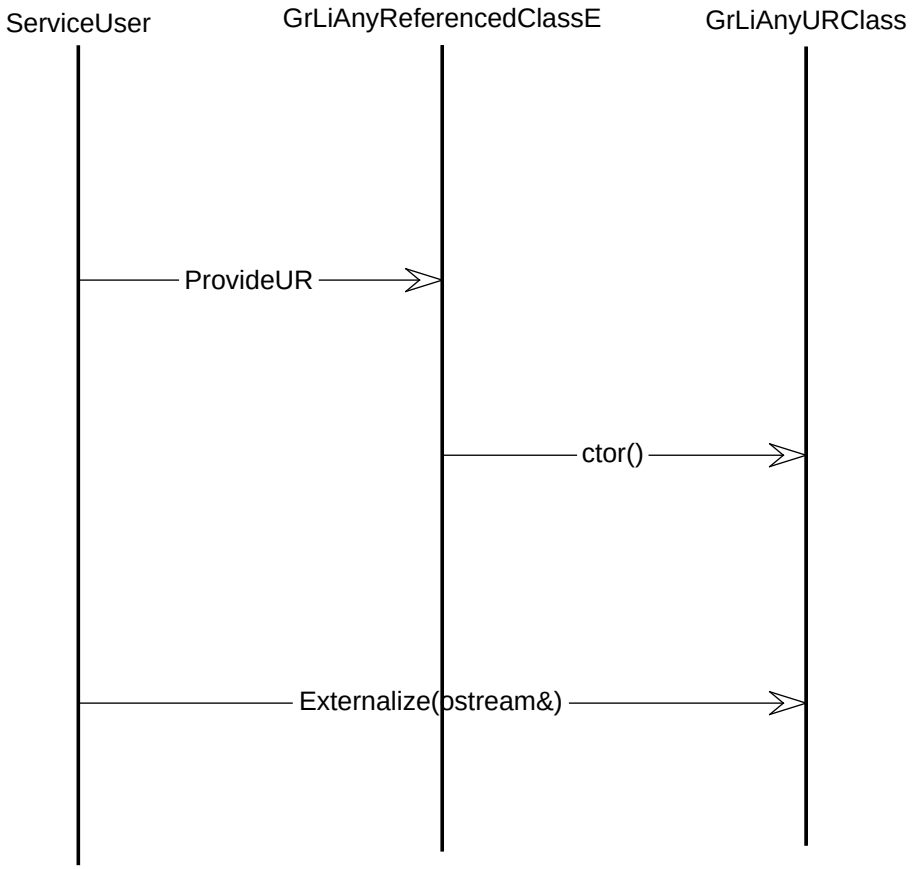

Figure 4.5.4.5-1 Event Trace for Creating and Externalizing a UR

4.5.4.5.2 Scenario #2

4.5.4.5.2.1 Abstract

This scenario describes going from the external representation of an UR to create the object which the UR references.

4.5.4.5.2.2 Interfaces

A client application shown as a ServiceUser object.

4.5.4.5.2.3 Stimulus

An application is required to reconstitute an object from an externalized UR.

4.5.4.5.2.4 Desired Response

The application gets back the originally created object.

4.5.4.5.2.5 Participating Classes

GrLiAnyURClass

GrLiAnyReferencedClass

UrMaker

UrProviderMaker

4.5.4.5.2.6 Pre-conditions

A UrMaker and a UrProviderMaker have already been instantiated.

4.5.4.5.2.7 Post-conditions

None.

4.5.4.5.2.8 Scenario Description

As with scenario 1, the GrLiAnyUrReferencedClass can be substituted in the scenario with session classes, data granule classes, and advertisement classes, among others, which will all be among those specialized from EcUrURProvider. The GrLiAnyUrClass can also be substituted with objects from session UR classes, data granule UR classes, and advertisement UR classes appropriate to the type of UR provider.

In this scenario, the ServiceUser object must restore the GrLiAnyReferencedClass object from the input stream which should carry the information necessary to rebuild the UR. The ServiceUser object calls the method MakeUr(istream&) with the input stream, which creates the UR object GrLiAnyUrClass in a two step process. First, the object GrLiAnyUrClass is constructed using a factory. Then it is filled with the data of the input stream using the method Internalize(istream&).

Now the ServiceUser is ready to reconstitute the object identified by the UR. The ServiceUser calls UrProviderMaker passing it the UR object created in the previous step. URProviderMaker uses the information in the UR object to construct a GrLiAnyReferencedClass object. UrProviderMaker first calls the constructor for a GrLiAnyReferencedClass, then requests a GrLiAnyReferencedClass to “Reconstitute” itself, passing it the UR object. Now an object exactly like the initial object that the UR stream was made from is rebuilt in memory.

4.5.4.5.2.9 Event Trace

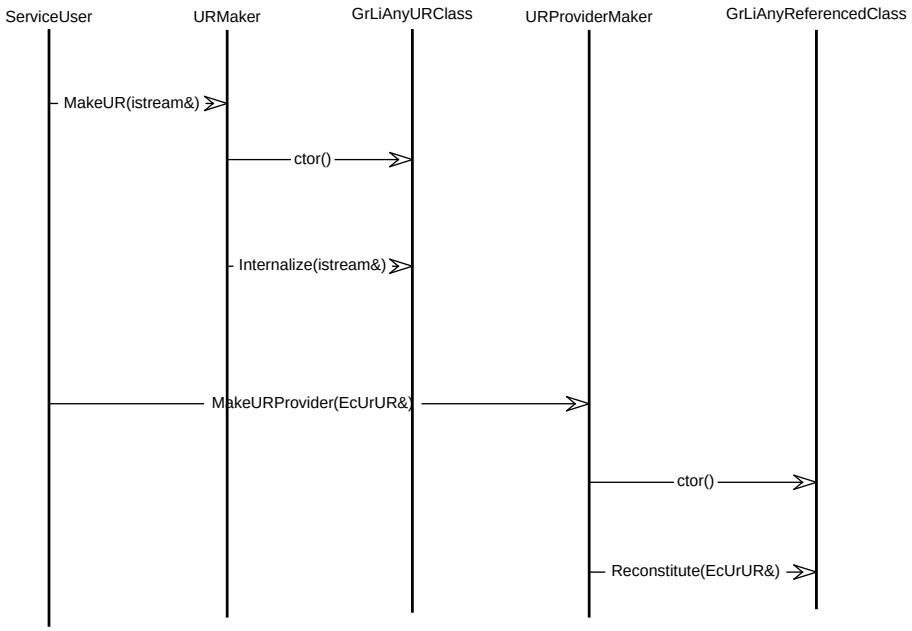

Figure 4.5.4.5-2. Event Trace for Internalizing and Reconstituting a UR

4.5.3.6 Implementation

The UR framework is a core infrastructural service and does not depend on any other CSCIs in the ECS system.

4.5.3.7 Service/CSCI Management and Operation

4.5.3.7.1 System Management and Strategy

The Universal Reference framework comprises of only one CSS (DCCI) CSCs, namely Univeral Reference.

4.5.3.7.2 Operator Interface

Not applicable.

4.5.3.7.3 Reports

None