

---

# Client Server Development

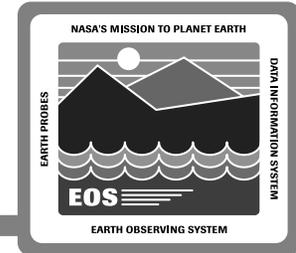
Aarathi

Laks

---

Developers Workshop  
30 May 1995

# Developing the Distributed Application



1

## Understand the Problem

- what needs to be distributed /resources
- think about layers & idempotent operations
- thread safety ex: X11/xt/Motif
- what are the security requirements
- understand the acknowledgments

2

## Define the Interface - IDL

- every Interface needs unique transaction
- parameters and results types
- Use ACFs

3

## Write the Server

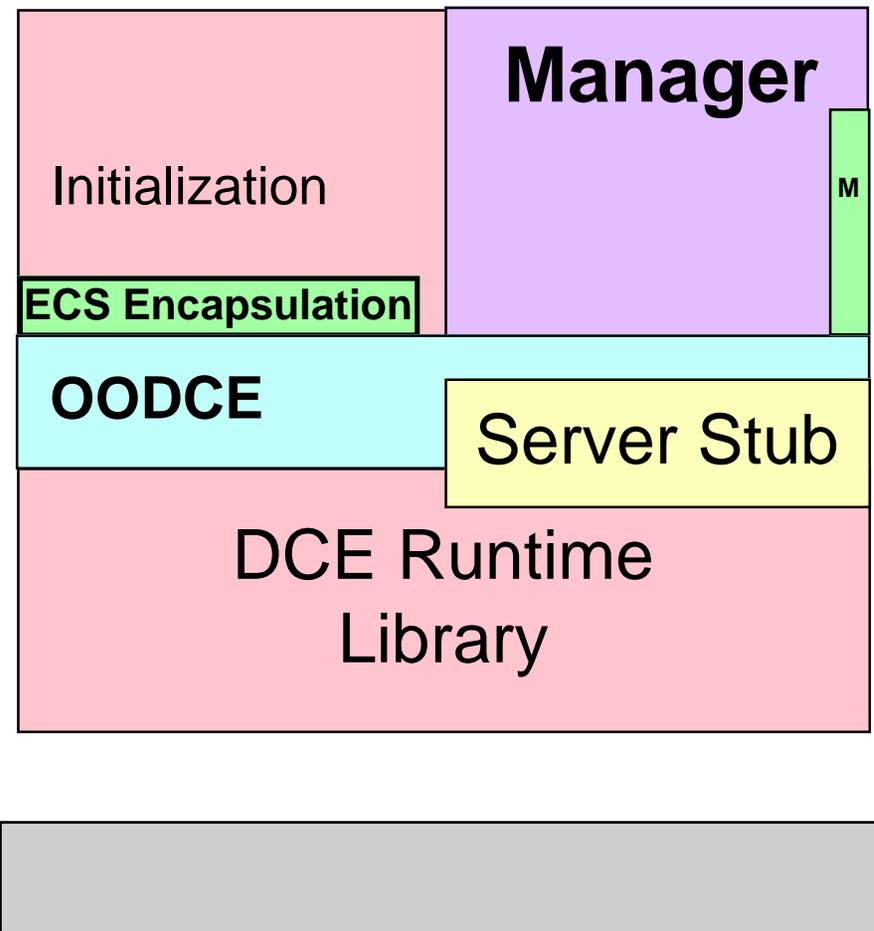
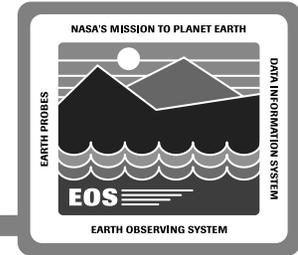
- Initialization & Registration
- Manager routines
- MSS interfaces

4

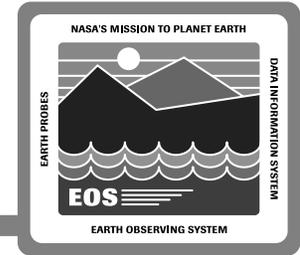
## Write the Client

- understand the use of threads
- Select the Binding method
- Invoke the rpc/member function

# ECS Server Basic Components

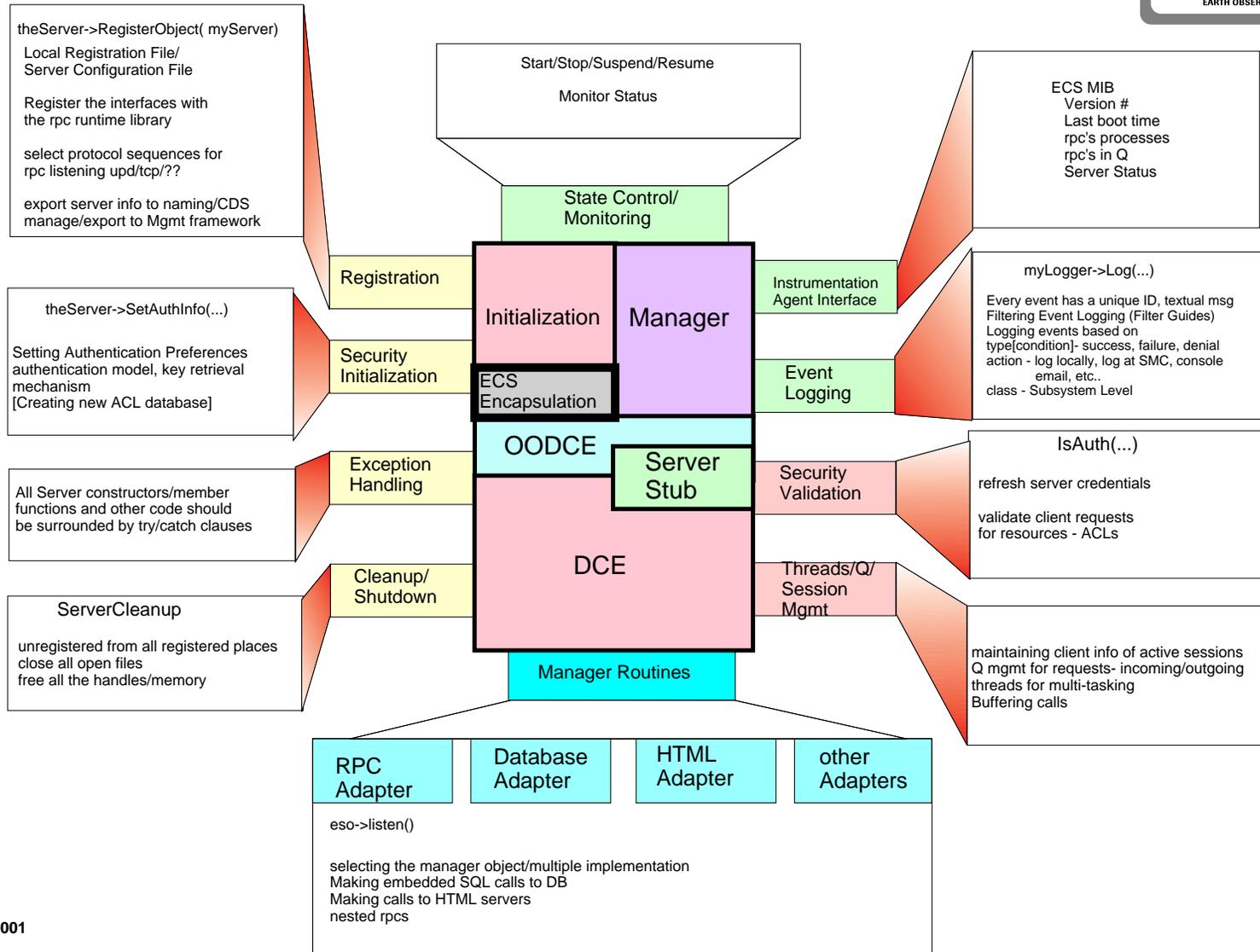
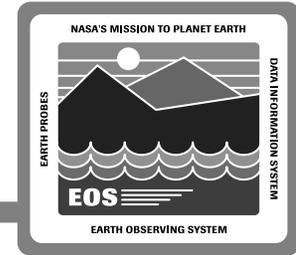


# What do ECS Server do

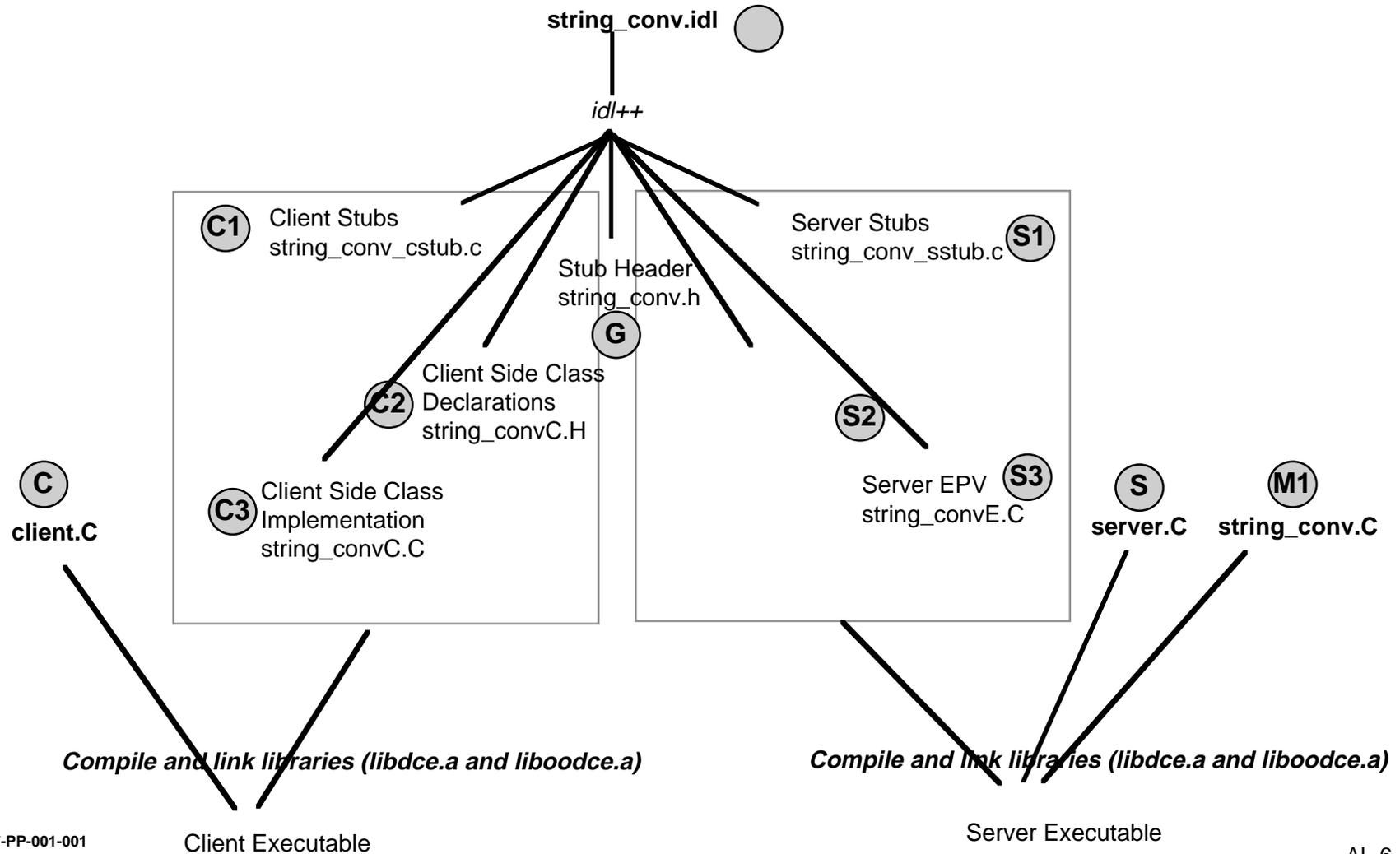
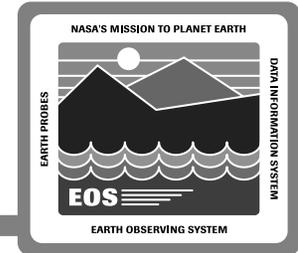


- ① Perform initialization and registration functions
- ② Block Waiting for incoming RPC request from a client
- ③ When a request arrives execute appropriate manager routine
- ④ Wait for more rpcs or management interface calls
- ⑤ Issue management/event notification
- ⑥ Do replicate update [optional]
- Exit gracefully, if necessary

# Generic ECS Server

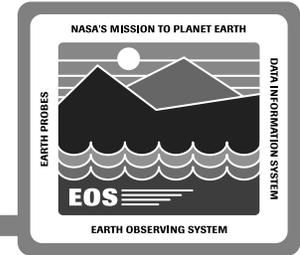


# OODCE



# Server Include Files

---



**Pthread.H**

**EcsServer.H**

**Exception.H**

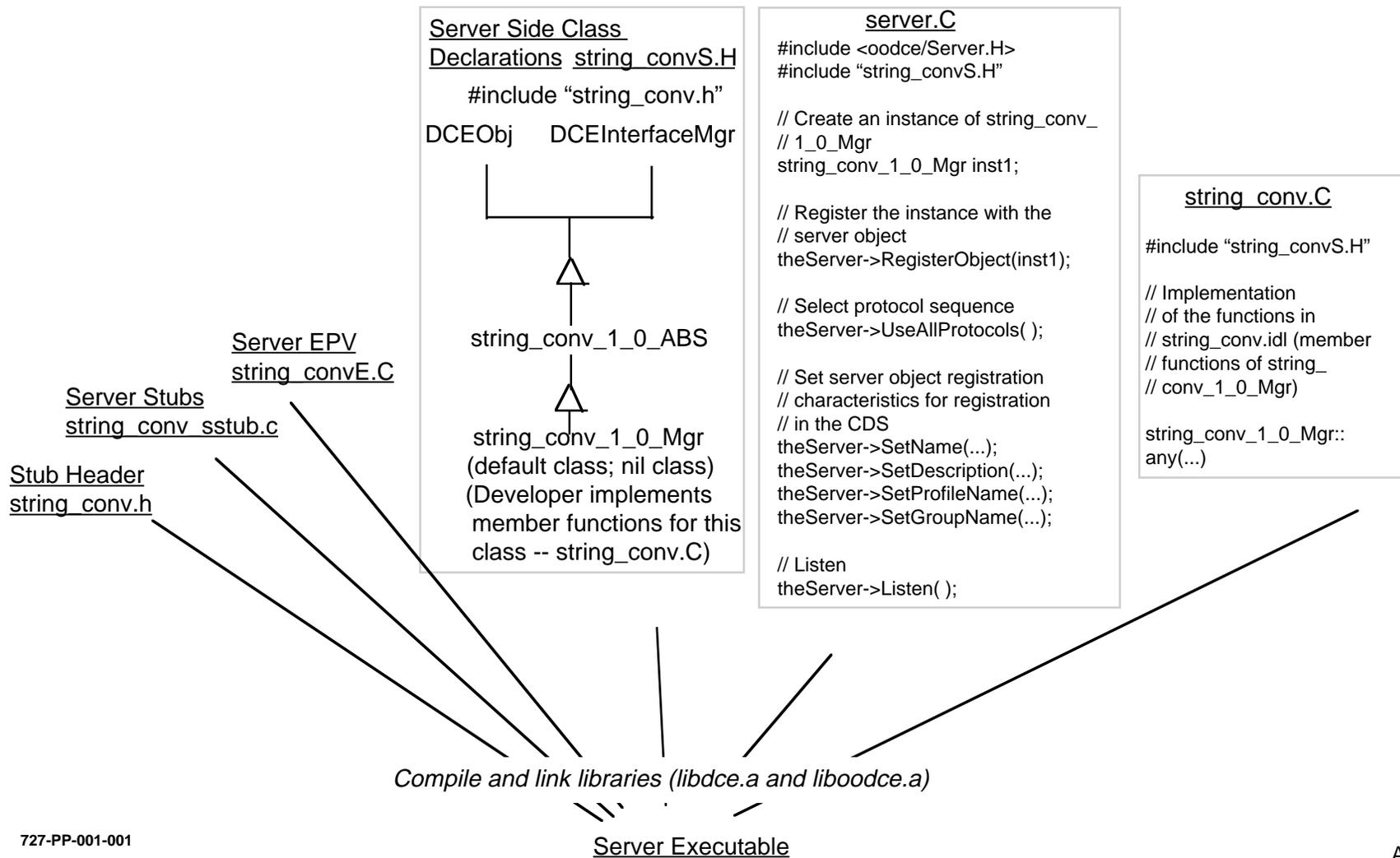
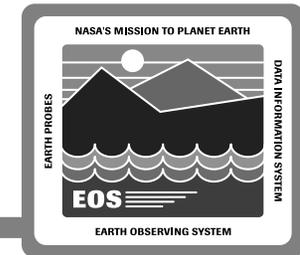
**Interface Header file**

**Roughwave header files**

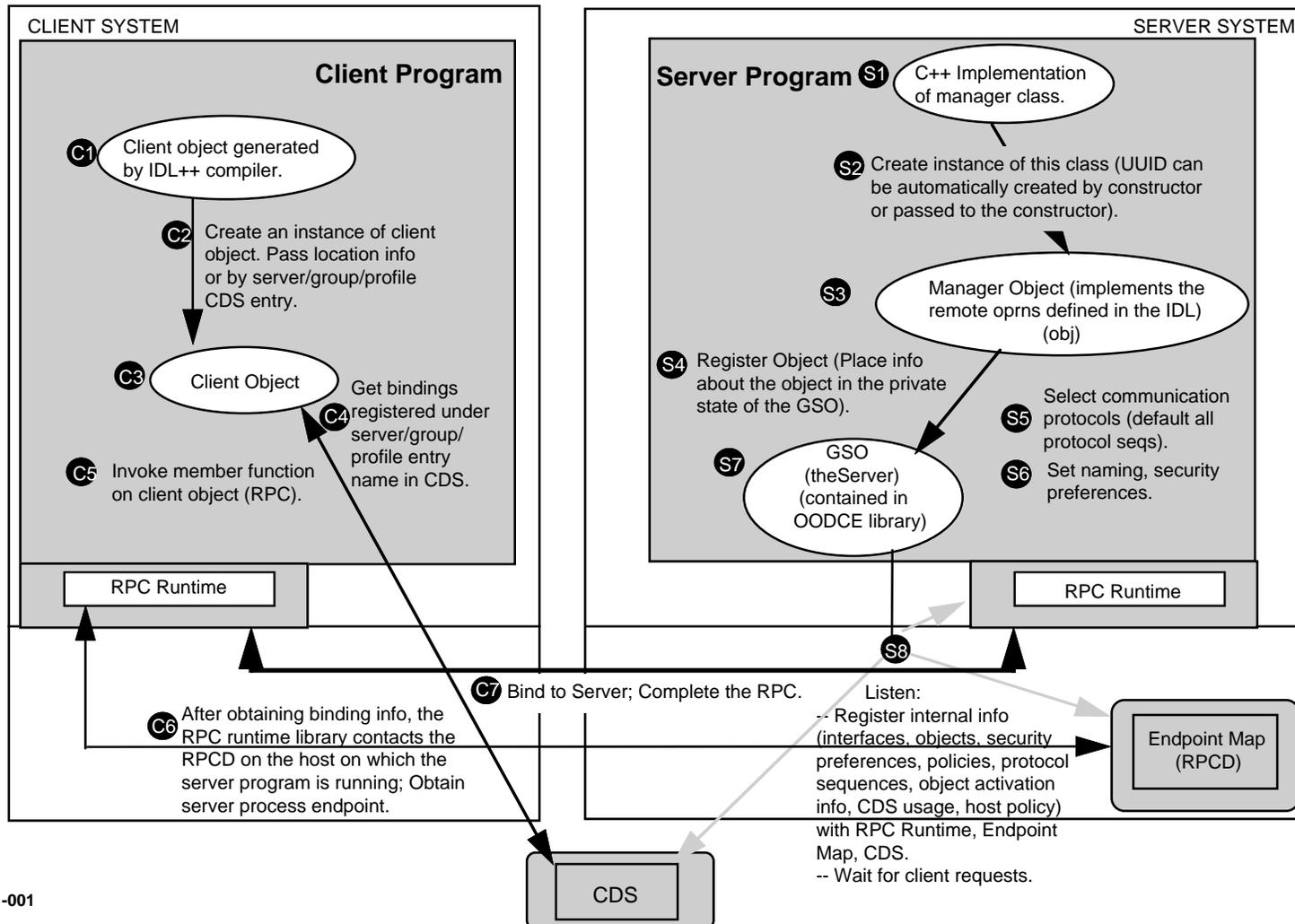
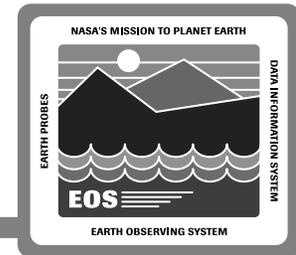
**ECS types file**

**Other application specific include files**

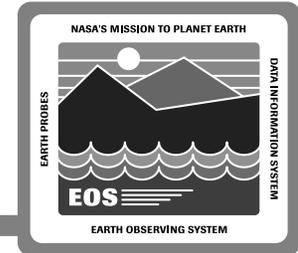
# OODCE Server Development



# Client Server in OODCE



# string\_conv.idl



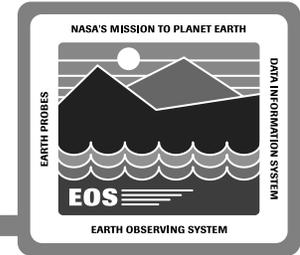
```
[uuid(59EDFC40-55F3-11CC-8C79-080009253B97),
 pointer_default(ref), /* for efficiency reasons */
 version(1.0)]
interface string_conv
{
 /* Define the data type to be used to hold the string data. */

 typedef [string]      char      string_ref_t[80];

 /*
 * Define the procedure to convert a string to uppercase. The string_convert_uppercase procedure takes a handle.
 * The other argument is a string data type used both for input and output. This emulates a C "call by reference" upon
 * return from the procedure call the pointer now points to different data, but the pointer address is still the same.
 *
 * The string_convert_uppercase function is declared to be idempotent for efficiency reasons. The idempotent attribute
 * tells the IDL compiler that if the function is called multiple times with the same arguments it will have the same effect.
 * This means a call to an idempotent function can be resent without adverse side effects; this allows the RPC protocol
 * to be more efficient, at least when using a datagram transport.
 */

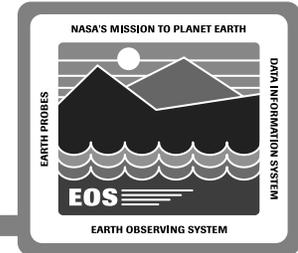
 [idempotent] void string_convert_uppercase
 (
     [in]      handle_t      h,          /* Use explicit binding */
     [in,out] string_ref_t  conv_string /* The string to convert */
 );
}
```

# Steps in developing a server main function



1. Construct manager objects that are accessed via the server.
2. Create and activate a signal handling thread to perform server cleanup.
3. Register objects created in step 1 with the Global Server Object (GSO).  
The GSO needs to know about all the objects it is managing.  
Registration places information about the object in the private state of the GSO.
4. Select communication protocols (optional).
5. Set naming preferences for the GSO (optional). Binding information can be placed in the CDS by giving GSO a name.
6. Set security preferences for the GSO (optional).
7. Instruct the GSO to listen for client requests. Places information with the Endpoint Map (RPCD) and CDS.

# server.C



```
#include <oodce/Pthread.H>           // DCEPthread Classes
#include <oodce/Server.H>           // DCEServer object definitions
#include <oodce/Exceptions.H>       // C++ Exceptions for library
#include <oodce/ObjectReference.H>   // Network location information for a manager object
#include "string_convS.H"           // DCEInterface definition
#include <stream.h>                 // C++ Streams

void main( )
{
    try { // Setup try block for exception handling
        // Create and activate a signal handling thread to perform server cleanup
        DCEPthread* exitThd = new DCEPthread(DCEServer::ServerCleanup, (void *) (0));

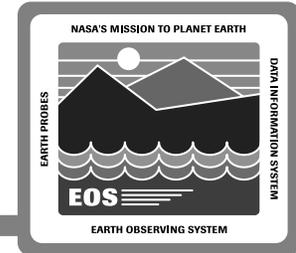
        // Create string_conv object with a given UUID
        DCEUuid objUuid = (const char *) "f47ee25c-480d-11ce-bb96-080009701906";
        string_conv_1_0_Mgr* string_conv = new string_conv_1_0_Mgr((uuid_t *) objUuid);

        // Register string_conv object with the server object
        theServer->RegisterObject(*string_conv);
        theServer->Register( );

        // Use all available protocol sequence for receiving client requests.
        theServer->UseAllProtocols( );

        // Set server object registration characteristics for registration in the CDS
        theServer->SetGroupName((const unsigned char *) "./subsys/HP/sample-apps/stringConvGroup");
        theServer->SetProfileName((const unsigned char *) "./subsys/HP/sample-apps/stringConvProfile");
        theServer->SetName((const unsigned char *) "./subsys/HP/sample-apps/stringConvServer");
        theServer->SetDescription((char *) "String Convert");
    }
}
```

# server.C (contd)



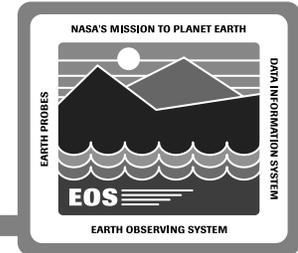
```
// For printing purposes
DCEBinding bindingHandle((rpc_binding_handle_t)string_conv->GetObjectReference( ));
printf("String Binding: %s\n", (unsigned char *)bindingHandle);

// Activate the server
theServer->Listen( );
}

// Catch any DCE related errors and print out a informative string if any occur
catch (DCEErr& exc) {
    cout << "Caught DCE DCEException\n" << (const char*)exc;
    throw;
}

// Destructors are called to do appropriate cleanup with DCE runtime and rpcd
}
```

# string\_conv.C

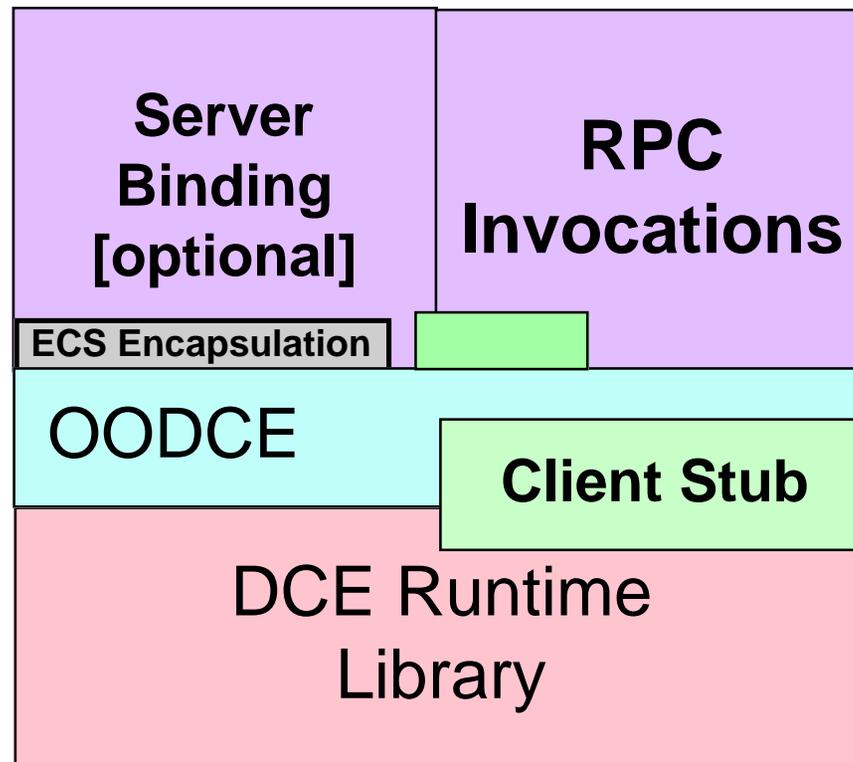
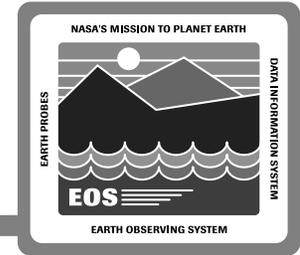


```
#include <stdlib.h>           // Standard POSIX defines
#include <stdio.h>           // Standard IO library
#include <ctype.h>           // For toupper()
#include "string_convS.H"    // From string_conv.idl

void string_conv_1_0_Mgr::string_convert_uppercase(string_ref_t conv_string)
{
    int i;

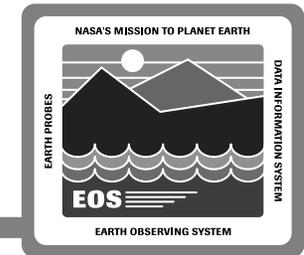
    printf("string_convert_uppercase method entered\n");
    for (i=0; (conv_string[i] != '\0') && (i < 80); i++)
    {
        conv_string[i] = toupper(conv_string[i]);
    }
    printf("string_convert_uppercase method exit\n");
    return;
}
```

# Basic ECS Client Components



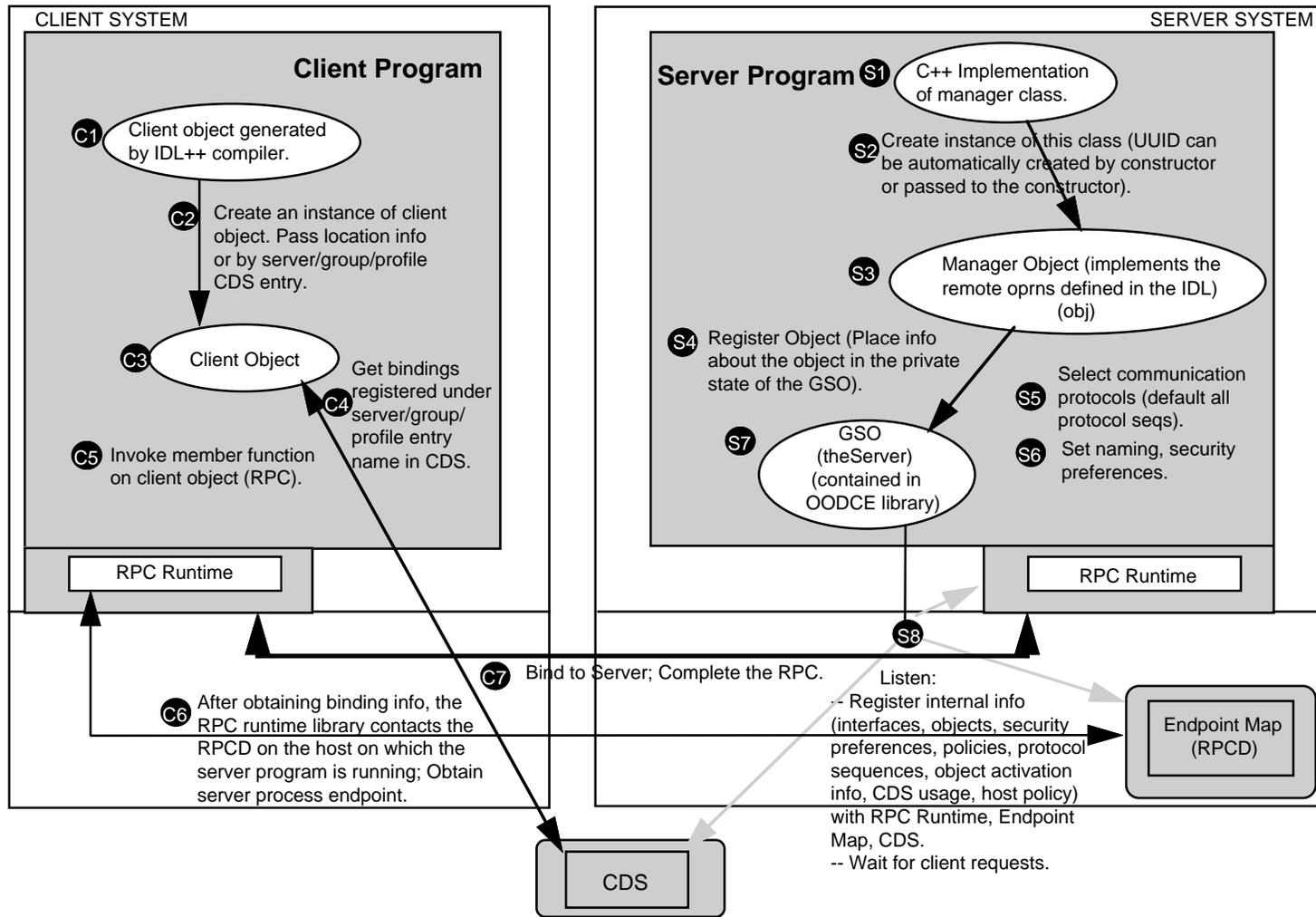
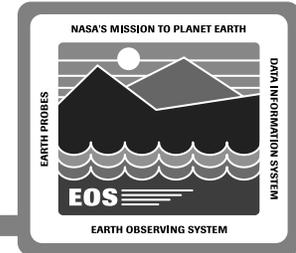


# Steps in developing client main function

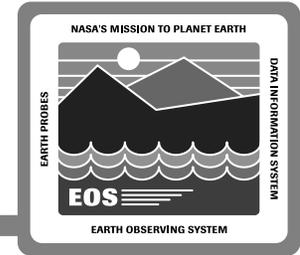


1. Create client object. The client object locates the manager object based on the following information:
  - Interface (default). Any object that implements the requested interface is used.
  - Host Address and Protocol Sequence
  - CDS
    - Server Entry
    - Group Entry
    - Profile Entry
  - Object Reference
2. Invoke member function on client object (RPC).

# Client Server in OODCE



# Binding Methods



## Automatic

- **Client Stub manages the binding handle**

## Implicit

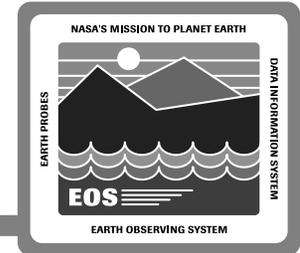
- **Client application obtains the binding handle but is held in the stub as a global variable**

## Explicit

- **Client application obtains it and is passed as the first argument of every rpc call [OODCE does this internally - developer need not worry]**
- **only method if the server supports more than one implementation of the same remote procedure calls using Object UUIDs**



# client.C



```
#include <oodce/ObjectReference.H>           // Network location information for a manager object
#include <oodce/Exceptions.H>               // Class library exceptions
#include "string_convC.H"                  // Client object definition
#include <stream.h>                        // C++ Streams

main( )
{
    try { // Set up a try block to catch exceptions
        unsigned char string_to_convert[80];
        strcpy((char *)string_to_convert, "asdf");

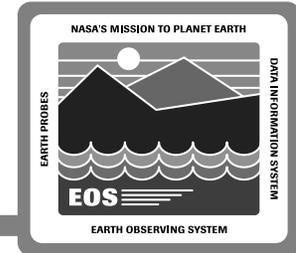
        cout << "Using Interface (the environment variable RPC_DEFAULT_ENTRY is set to ../subsys/
        HP/sample-apps/stringConvServer)" << endl;
        string_conv_1_0 convert0;
        convert0.string_convert_uppercase(string_to_convert);

        cout << "Using Host Address and Protocol Sequence" << endl;
        string_conv_1_0 convert1((unsigned char *)"baltic.hitc.com", (unsigned char *)"ncadg_ip_udp");
        convert1.string_convert_uppercase(string_to_convert);

        cout << "Using CDS Server Entry Name" << endl;
        string_conv_1_0 convert2((unsigned char *)"../subsys/HP/sample-apps/stringConvServer");
        convert2.string_convert_uppercase(string_to_convert);

        cout << "Using CDS Group Entry Name" << endl;
        string_conv_1_0 convert3((unsigned char *)"../subsys/HP/sample-apps/stringConvGroup");
        convert3.string_convert_uppercase(string_to_convert);
    }
}
```

# client.C (contd)



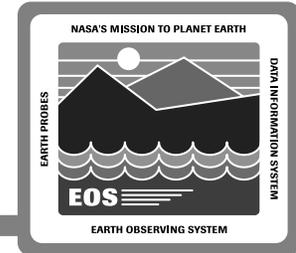
```
cout << "Using CDS Profile Entry Name" << endl;
      string_conv_1_0 convert4((unsigned char *)"/./subsys/HP/sample-apps/stringConvProfile");
      convert4.string_convert_uppercase(string_to_convert);

cout << "Using Object Reference" << endl;
DCEBinding bindingHandle((const unsigned char *)
    "f47ee25c-480d-11ce-bb96-080009701906@ncadg_ip_udp:155.157.31.90[]");

DCEObjectReference objRef(bindingHandle);
objRef.PrintReference( );
string_conv_1_0 convert5( (DCEObjRefT *)objRef );
convert5.string_convert_uppercase(string_to_convert);
}

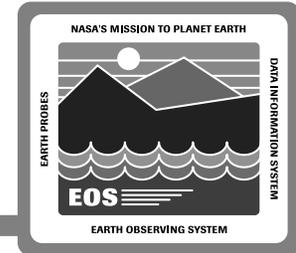
// Catch any DCE related errors and print out a informative string if any occur
catch (DCEErr& exc) {
    printf("DCE DCEException: %s\n", (const char *)exc);
}
}
```

# Glossary



|                               |   |
|-------------------------------|---|
| Binding Information           | Includes one or more sets of protocol sequences and host address combinations. Well-known endpoints can be part of the binding information but dynamic endpoints cannot. This is the information a client needs in order to find a server.  |
| Cell Directory Service        | The Cell Directory Service (CDS) is a name service supplied with DCE which is used by applications to store and retrieve binding information.   |
| Endpoint                      | An endpoint is a number representing a specific server process running on a system.   |
| Endpoint Map                  | Server process places process information in a special database on the server system called the local endpoint map.   |
| Entry Point Vector            | An Entry Point Vector (EPV) contains entry points for each RPC defined in an interface. In DCE, these are pointers to remote procedures and in OODCE these are pointers to member functions of C++ objects. The EPV code ensures that the correct C++ manager object on the server is called for each client request.   |
| Global Server Object          | A Global Server Object (GSO) called theServer is used to register objects with the DCE environment and then to listen for client requests. OODCE library contains the GSO.  |
| Group Entry                   | A group entry is a name service entry that corresponds to a set of servers, usually offering the same interface.<br>The name service routines search the members of a group to find a server.   |
| Interfaces                    | Operations that can be performed on a DCE object are grouped into logical sets called interfaces. DCE Interfaces defines the calling syntax that is used by both the requestor (DCE client) and the provider (DCE object) of an operation.  |
| Interface Definition Language | Specification of interfaces use an Interface Definition Language (IDL) to define the operation signatures.  |
| Interface Identifier          | During the search for binding information, RPC name service routines use this identifier to determine if a compatible interface is found.   |
| Interface Versions            | Interface versions are identified by a major and minor number. An increase in the minor number signifies a compatible upgrade to the interface. Interfaces with different major numbers are not compatible.   |
| Marshalling                   | Marshalling is the process during a RPC which prepares data for transmission across the network. Marshalling converts data into a byte-representation format and packages it for transmission using a network data representation (NDR). NDR handles differences like big-endian versus little-endian (byte order), ASCII characters versus EBCDIC characters, and other incompatibilities. |
| Object                        | An OODCE object is an entity that is manipulated by a set of well defined operations. A server can manage thousands of objects. These objects are accessed via a server process that is said to export interface information on behalf of the object it supports.   |
| Object UUID                   | Each OODCE Object is assigned a UUID so that it can be located and used by remote clients.  |

# Glossary (contd)



|                     |   |
|---------------------|---|
| Profile Entry       | A profile entry is a name service entry that defines a search list for finding servers in the CDS. Profiles gather all services together. Profiles let you tailor the CDS search so that all your clients begins a search from a single general entry name. |
| RPC Daemon          | The RPC daemon (RPCD) is a process that provides the endpoint map service. This service maintains the local endpoint map for local RPC servers and lookup endpoints for RPC clients.  |
| RPC Runtime library | The RPC Runtime library is a set of standard runtime routines that support all DCE RPC applications. The client stub communicates with the server stub using the RPC runtime library.   |
| Server Entry        | Server entry stores binding information for an RPC server. It contains interface identifier, binding information and an optional object UUID.   |
| Stub                | Stub is a surrogate code that supports remote procedure calls.  |
| Unmarshalling       | Data transmitted across the network undergoes a process called unmarshalling. If the data format of sender and receiver is different, the receiver's stub converts the data to the right format for that system, and passes the data to the application.    |