

The cover art features a dark blue header with a white oval on the left. Below the header is a large, abstract image of a person in a dynamic, athletic pose, rendered in a glowing, translucent style against a warm, orange-brown background. The text 'Roma BSP' is prominently displayed in a large, metallic, 3D font on the left side. The 'Candle' logo is visible in the bottom right corner of the image area.

Roma Business Service Platform

!Candle®

**COM/CORBA
Interoperability &
Roma Access for
COM/CORBA**

**Consulting &
Services**

August 1999

**Michael S. Pallos
Solution Architect
Candle Corporation
2425 Olympic Boulevard
Santa Monica, California 90404
(770) 437-7697**

Published by

!Candle[®]

Statement of Confidentiality

The information contained in this document is confidential and proprietary. In no event shall all or any portion of this white paper be disclosed or disseminated without the express written permission of Candle Corporation.

© 1999 Candle Corporation. All rights reserved, international copyright secured.

Table of Contents

| | |
|---|-----------|
| 1. OBJECT REQUEST BROKER | 1 |
| INTRODUCTION | 1 |
| <i>Protocol Independence</i> | 1 |
| <i>Interface Independence</i> | 1 |
| INTERFACES | 2 |
| <i>Interface Definition Languages</i> | 2 |
| <i>Skeletons and Stubs</i> | 2 |
| PROTOCOL SUPPORT | 3 |
| <i>Object Identification</i> | 3 |
| <i>Object Activation</i> | 3 |
| <i>Multi-threading Support</i> | 4 |
| SUMMARY | 4 |
| 2. COMPONENT OBJECT MODEL | 5 |
| HISTORY OF COM | 5 |
| COM ARCHITECTURE | 6 |
| INTERFACES AND OBJECT IDENTITY | 7 |
| <i>Component Activation</i> | 7 |
| <i>Protocol Support</i> | 8 |
| <i>Management of Object Lifecycle</i> | 8 |
| 3. COMMON OBJECT REQUEST BROKER ARCHITECTURE | 9 |
| HISTORY OF CORBA | 9 |
| <i>The OMG Technology Adoption Process</i> | 9 |
| CORBA OBJECT REQUEST BROKER | 10 |
| <i>CORBA IDL</i> | 10 |
| <i>CORBA Protocol Support</i> | 10 |
| <i>Dynamic Interfacing with CORBA - DDI and DSI</i> | 11 |
| CORBA OBJECT LIFE CYCLE | 12 |
| 4. ROMA COM/CORBA ADAPTER SPECIFICATIONS..... | 14 |
| OVERVIEW | 14 |
| <i>Platform Coverage</i> | 14 |
| HIGH LEVEL ARCHITECTURE | 15 |
| <i>ObjectBridge Architecture</i> | 15 |
| <i>Roma Adapter Architecture</i> | 15 |
| <i>Interface Repository Architecture</i> | 15 |
| LOCAL REPOSITORY ISSUES | 16 |
| BROWSING THE ROMA ADAPTER | 17 |
| SYNCHRONOUS REPLIES | 17 |
| 5. ROMA'S EXTERNAL SPECIFICATIONS..... | 18 |

| | |
|--|-----------|
| OVERVIEW | 18 |
| LDAP REPOSITORY | 18 |
| <i>Overview</i> | 18 |
| <i>Development Work</i> | 18 |
| ASYNCHRONOUS METHOD (CALLS INTO ROMA) | 18 |
| <i>Exposure</i> | 18 |
| CORBA & COM SERVERS (CALLS FROM ROMA) | 19 |
| <i>Exposure</i> | 19 |
| <i>Exposure example</i> | 20 |
| <i>Technical Overview</i> | 21 |
| 6. ROMA BSP OBJECT ACCESS FOR COM/CORBA | 22 |
| PREREQUISITES | 22 |
| <i>Introduction</i> | 22 |
| <i>Operating System</i> | 22 |
| <i>Communications</i> | 22 |
| <i>Additional Roma Software</i> | 22 |
| <i>Supported Object Systems</i> | 22 |
| <i>Messaging Software and Configuration</i> | 23 |
| INTRODUCTION | 23 |
| <i>Purpose of This Release</i> | 23 |
| <i>What is Roma Object Access?</i> | 23 |
| <i>Features</i> | 23 |
| <i>Roma Object Access Browser</i> | 25 |
| <i>Roma Object Access Agent</i> | 25 |
| <i>Roma Object Access LogView</i> | 25 |
| OVERVIEW OF FEATURES IN ROMA BSP 3.0..... | 25 |
| <i>Additional features in Roma Object Access for Roma BSP 3.0 (Phase II)</i> | 25 |

1. Object Request Broker

Introduction

This goal of this paper is to provide an initial understanding to the two leading object request brokers currently available in our industry. The Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) and the Common Object Model (COM) from Microsoft.

Prior to discussing the differences between CORBA and COM, we will first focus on the simulates. Generally speaking a distributed object backbone must have two properties; a) Protocol Independence and b) Interface Independence.

Protocol Independence

The protocol must be independent of the platform and the component environment. Remote components require a higher-level protocol that the standard networking protocols, such as TCP/IP. General approaches include sitting the higher-level protocol on industry standard network protocols such as TCP, UDP, and others.

Interface Independence

There must be interface independence when attaching the object to the backbone, (ORB). The interface approach includes the utilization of a powerful interfacing technology - Interface Definition Language.

This allows the application or object to be development in multiple languages and the 'connecting' to the ORB is handled by the IDL, as seen in Figure 1.0

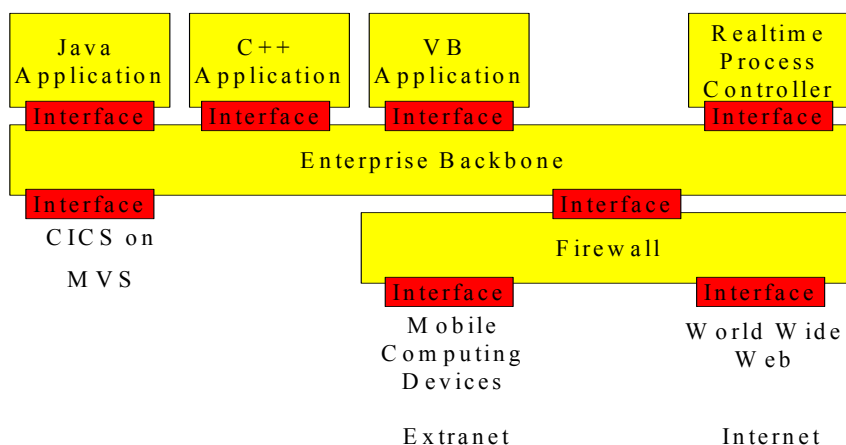


Figure 1.0 - Enterprise System Example

Interfaces

Interfaces allow the components to be designed as black boxes. Self contained pieces of logic that performed specific tasks. The language and operating system they are utilizing is not relevant. The goal is to allow the developer to utilize which every development language and operating system that is most strategic to the corporation. The 'plugging in' or connecting to the ORB backbone is the interfaces responsibility. This flexibility allows the enterprise to maximize their current infrastructure and skill set, while providing a mechanism for connecting all of the black boxes.

Interface Definition Languages

Interface Definition Languages provides what many consider to be the most important ORB feature. The IDL provide a language neutral method of documenting interfaces for the components. This documenting is suitable for use by the compiler. Much like the preprocessor in C. The IDL processor takes the neutral definition language document and converts into the target language producing a skeleton and stub which will be detailed in the following section. Providing interface object code to be linking in with the component build process.

IDL is purely declarative. Interfaces can be defined with regard to; parameters, operations or return types. They can not define application functionality. There is also not conditional functionality allowed in the IDL such as, IF, ELSE, and looping conditions.

Skeletons and Stubs

ORBs provide a compiler which convert the neutral IDL into a target language. Two object files are generated. One for the client (called the stub) and one for the server (called the skeleton). After running the IDL through the compiler, a native language support library is generated. The object file is then linked in during the build process. The responsibility of connecting the client to the server, or stub to the skeleton is contained with the stub/skeleton object file removing the complexities of the interface from the developer.

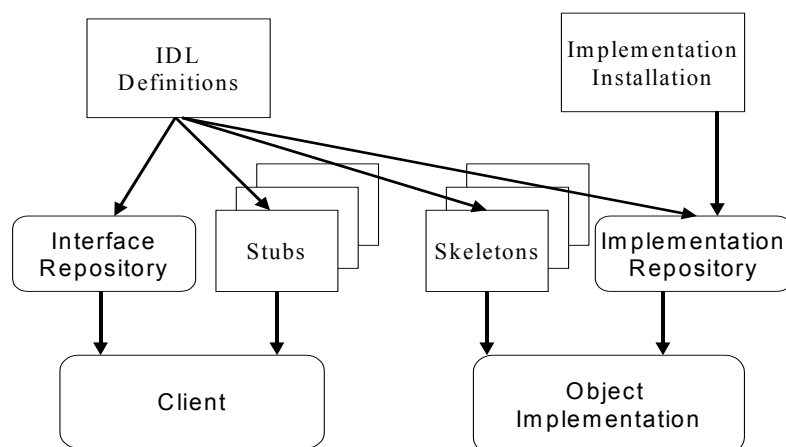


Figure 1.2 IDL Compiler Output

Protocol Support

Object Request Brokers must support a common higher-level protocol. The protocol must be capable of locating the target object, providing information about the target interface, handling support for advance services, and exchanging data.

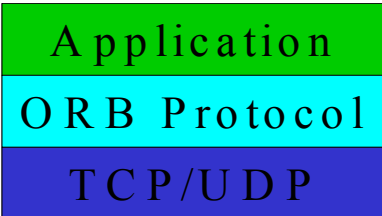


Figure 1.3 - CORBA Protocol Stack

Object Identification

Object Identification is one area in which CORBA and COM are substantially different. The application must be able to locate an object, load it into memory on a remote server, and have a unique reference allowing calls to be made. COM and CORBA object identification is detailed later in this paper.

Object Activation

When an application requires the service of a component on the ORB, the ORB will locate and activate the object on behalf of the requesting client. This activation generally uses the well-known port method or a broadcast-based discovery method for finding the activator.

Using the well-known port method, an activator process will be started at system boot. The activator will be allocated a well-known TCP port. The server listens for incoming connections on this port. The port acts as the initial target location who then responds with additional information containing the destination of the port required for the client. Along the lines of a port traffic cop, or the port broker.

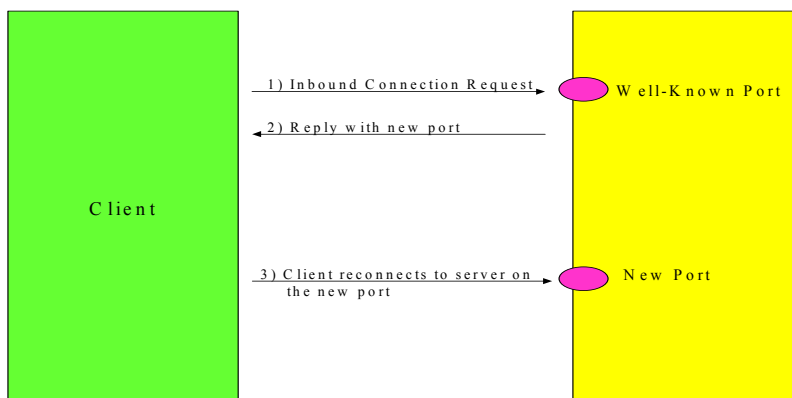


Figure 1.4 - Connecting to a well-known port.

Another common model is to utilize the UDP to broadcast an initial request. The UDP message contains the client information, the name of the server, and other connection parameters. This information is received by either an activation component or possibly the object server. Each server that supports the required object will reply unless other rules are specified. The client uses the list of replies to connect to a suitable implementation.

Multi-threading Support

A thread can be viewed as a lightweight process within a containing parent process. Generally, server-based application in the past have used the process per connection method for load-balancing between incoming requests.

Threads provide a more flexible model for this type of processing. Since a thread inherits the parent environment and new thread initialization is substantially more efficient than starting a new process, it is an ideal model for application servers.

Summary

Object request brokers must provide application interface functionality and a transport to allow transmission of request/reply sequences between objects. Interface Definition Language supports the application interfacing as well as the second approach using a dynamic interfacing mechanism.

The ORB must provide a higher-level transport riding above the network protocol.

2. Component Object Model

Microsoft defines its distributed Component Object Model (COM) as "An object protocol that enables ActiveX components to communicate directly with each other across a network. COM is language-neutral, so any language that produces ActiveX components can also produce COM applications."

The vast majority of GUI development tools for the Windows platform provide support for Microsoft ActiveX. Microsoft Windows is delivered with support for interprocess communication using Microsoft COM. Microsoft COM is optimal for building single-machine applications that need to communicate with other application components on the same machine, or within the constraints of a Microsoft environment.

History of COM

Microsoft ActiveX began as a mechanism to support the concept of compound documents. Compound documents present data from different applications as a single document. This was intended primarily for Microsoft Office. User could take a PowerPoint slide and insert it in a Word document, or embed an Excel spreadsheet in a PowerPoint presentation.

The first incarnation was termed Object Linking and Embedding (OLE). Objects not in the sense of object programming, but as references to document objects. The contained objects could either be linked or embedded into the container. Although OLE was created with the end-user in mind, Microsoft successfully extended it to the common programming issue; How to get maximum reuse out of code within and across platforms. The result was OLE 2, which introduced the concept of a software backbone, the Common Object Model (COM). COM introduced a standard method for components to communicate.

In parallel with the document containers, Visual Basic started to extend the use of GUI components to the basic toolkit. These add-on components were known as Visual Basic Extensions (VBX). VBX controls were based on the 16-bit environment. They became outdate with the release of 32-bit Microsoft platforms.

Microsoft introduced the new OLE Control Extensions (OCX) that provided 32-bit replacement for the VBXs. Unlike VBX, Microsoft based these on the new OLE COM Standard. Since OCX is based on Microsoft COM, a basic blueprint was available for developers to build OCX controls and also for third-party development tools to provide support. Thus producing a melding of both Microsoft worlds, OLE and VBX, with a integrated, 32-bit environment, Microsoft standard.

OCX controls provide some interesting features. They are small pieces of executable content. They can communicate using a network protocol. They also contain the primary features required for Internet usage. Today, OCX controls are a type of Active/X component, as are other COM-enable components. COM is the glue that holds them all together.

Microsoft recently extended COM adding support for remote COM objects. This extended protocol allows COM applications to cooperate with remote components as if they are on the same processor.

ActiveX is very much Windows-centric, Microsoft is working with neutral vendors such as IONA Technologies and directory with UNIX vendors such as Digital Equipment Corporation and HP to ensure that ActiveX has a part to play on these platforms. (At least that is what Microsoft is telling to outside world.)

COM Architecture

ActiveX describes a family of related technologies. At the highest level in this family is the ActiveX component, which is simply an object that provides a COM interface, figure xxx.

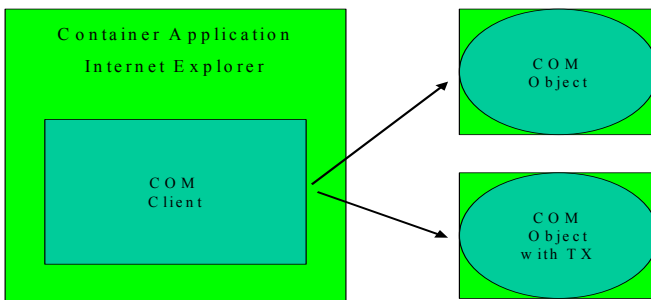


Figure 2.0 - Overview of ActiveX Family

The members of the ActiveX family and their interaction from a high level is reflected in Figure 2.0. At the highest level are COM objects. The remote COM object provides a COM interface. The COM interfaces exposes the internal object class to the local COM implementation. A variation of the COM is one that has transactional ability (identified with the TX) provided by the Microsoft Transaction Service. The third type of COM object illustrated is the ActiveX control running within the constraints of an application. Reflected above is COMs ability to provide a interoperable foundation in which ActiveX controls support the standard COM interface with extensions that allow for plug-and-play interoperability with other applications.

Each physical machine shown in Figure xxx supports a local COM runtime. The COM runtime is responsible for ensuring that calls by client applications are correctly directed to the required implementation object. Objects can either be local or remote. In the case of a local object, COM simply activates the component and allows direct calls. In the case of a remote object, calls are intercepted and passed using an RPC mechanism to the remote machine. At the remote machine, the local COM implementation will activate and delegate calls on behalf of the machine.

Fundamentally, Microsoft COM fits with the generic model of an object request broker. The logical architecture of COM is reflected in Figure xxx.

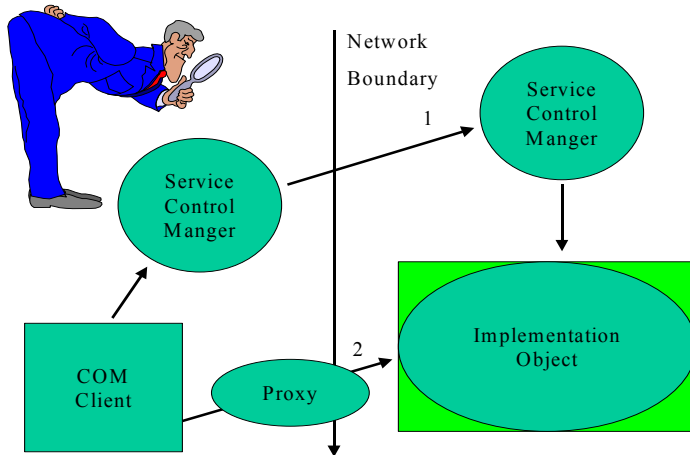


Figure .21 - Logical COM Architecture

The client is the component requesting a service, the component is the actual implementation. Although Figure xxx indicates a remote instance, the ActiveX architecture will handle local components, including loading these components into the same address space as the client.

The elements of the Service Control Manager (SCM) provide simple location and activation services. The protocol used for remote communication is based on DCE RPC. This provides a higher level protocol for the ORB. At a lower level, Microsoft COM supports TCP, UDP, and HTTP.

The object proxy at the client side is object-specific, although it will contain some standard COM support code. The proxy is responsible for the interception and marshalling of remote object calls. The stub provides the opposite function.

Interfaces and Object Identity

COM uses a GUID (Globally Unique Identifier) to identify object interfaces. The GUID concept was borrowed from the UUID concept by the Open Systems Foundation (OSF), Distributed Computing Environment (DCE). This is a 16-byte value that is guaranteed to be unique in "time and space". The algorithm uses the time of GUID creation and the network address of the machine and uses these two seed values to generate the 16-byte value. This information is stored in the Windows registry.

Component Activation

Information is contained within the Windows registry. An implementation registers itself by providing a unique class id, the type of server, and the location of the server's binary. Objects are implemented in servers and these can be of three kinds - a) in-process. b) local and c) remote.

Microsoft's variation of DCE RPC provides additional functionality that allows the activation of remote components. Previous implementations of RPC required that the server be activated prior to client connections. COM uses a similar module to a generic activator for object activation. It

listens for incoming connections on a well-known port. The Service Control Manager (SCM) is the COM activator. One SCM is available per machine hosting COM objects. The remote SCM implements the RemoteActivation interface. This interface provides a single method, RemoteActivation, which accepts an incoming CLSID. With the CLSID, the SCM can review the local registry and launch the object.

Protocol Support

The protocol used by Microsoft COM leverages work by OSFs DCE. DCE provides a point-to-point communication protocol, the Remote Procedure Call (RPC).

Management of Object Lifecycle

COM uses a reference counting mechanism to manage the life cycle of an object. COM maintains a count of each connection that is decreased when the client closes the connections. The interface provides methods for increasing and decreasing the reference count. When the reference reaches zero, the object is destroyed.

Using reference counting for object garbage collection requires programming discipline. Those of use familiar with C++ are aware of the problems caused by failure to call destructors in order to destroy an object. Java automatically handles object garbage collection. Once the object is out of scope the Java Virtual Machine marks it for deletion. Recently COM has been extended in order to prevent objects from early destruction or immortality. The COM smart proxy removes this overhead from the programmer.

Reference counting does have some drawbacks. Clients may have terminated without calling the release method and allowing the server to close the connection. COM uses a ping protocol for detection of dead clients. This is similar to the TCP/IP keep-alive mechanism. If after a total of three pings the client does not respond, the connection is closed.

The pinging mechanism is optimized. For example, if a client is connected to 100 components on a single server, a single ping will be sent to the client for keep-alive. This deactivation mechanism can be turned off for operation if the administrator believes he has a stable networking environment.

Once the reference count for an object reaches zero, the object is terminated. COM object references are pointers to the address in memory where the object is loaded. Therefore, once the object is terminated, the object reference is invalid. CORBA takes a different approach to object creation and destruction.

3. Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) provides multiple options allowing for integration with almost any type of application or environment. In order to be truly open the design of CORBA provides multiple possible usage models, static and dynamic.

History of CORBA

The Object Management Group is an international organization supported by over 800 members. The members include all of the major vendors of systems and software from around the world. The notable exception is Microsoft which has its own competing object broker called the Distributed Component Object Model (DCOM). One third of OMGs members come from outside the United States, most from Europe, but about 5 percent of the total from other countries, chiefly in the Asian Pacific Rim but also including Australia, Africa, and South America. Founded in 1989, the OMG promotes the use of object-oriented software development.

Unlike some other consortium, the OMG does not produce software, only specifications. The specifications are freely available for any company (OMG member or not) to implement; neither explicit permission nor fee is required. OMG expects that implementations will come from many companies.

The OMG Technology Adoption Process

The OMG works on the principle of task forces. When a certain technology are needs debate and a solution, the task for will issue a Request For Proposal (RFP). Members of the OMG will respond with a draft technical solution.

In 1990 the OMG first published the Object Management Architecture (OMA). This details the four major components of the CORBA infrastructure, as reflected in figure xxx.

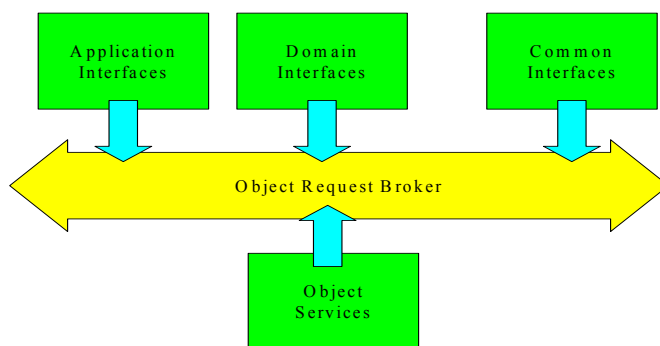


Figure 3.0 - Four major components of the CORBA infrastructure

The first of the components is the actual ORB that provides the interfacing and messaging infrastructure. CORBA and COM provide both basic application services. The CORBA services are an integral part of the OMA. The common facilities are predefined vertical or horizontal 'services.' Horizontal facilities are generally common to any application. For

example, systems management. Vertical facilities are specifications for industry-specific components and frameworks. For example, financials and banking. Finally, the application interfaces are the business objects.

CORBA Object Request Broker

In reference to the generic object model CORBA contains two key components, as reflected in Figure xxx. CORBA IDL allows component developers to define interfaces between the application and the ORB. CORBA IIOP allows components to communicate across network boundaries.

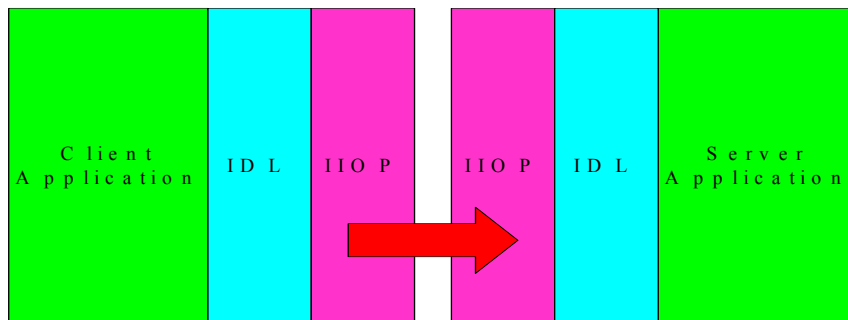


Figure 3.1 - CORBA Components

CORBA IDL

CORBA Interface Definition Language (IDL) provides a language-neutral method of defining interfaces between components. IDL is a purely declarative language, allowing definition of exposed data structures and object operations or methods. No support is provided for conditional processing, looping or other similar programming constructs.

IDL can also provide ORB control constructs, for example it will support the definition of one-way operation calls where the client will not block while waiting for a reply.

CORBA Protocol Support

All CORBA 2.0 specifies a mandatory protocol for inter-ORB communications. This protocol, based on a TCP/IP transport, is referred to as the Internet Inter-ORB Protocol (IIOP).

OMG also defined the Environment Specific Inter-ORB Protocols (ESIOPs). ESIOPs allow the extension of a standard ORB to support other protocols. The first ESIOP defined by the OMG was the Distributed Computing Environment Remote Procedural Call (DCE RPC). This allows CORBA-compliant ORBs to bridge between DCE and CORBA environments. The protocol stack is reflected in Figure xxx.

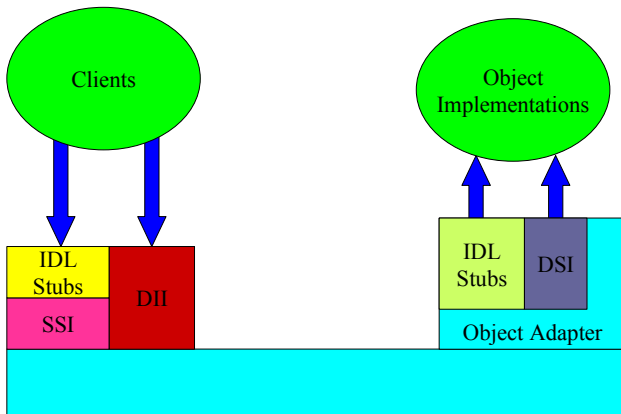


Figure 3.3 - Dynamic and Static Combinations for Request Handling

DII and DSI play an important part in bridging to systems when static IDL is inappropriate. Uses include bridging to a 4GL language, integration with legacy system, and bridging between object models.

CORBA provides an interface repository that acts as a type repository for stored interface information. An interface that can be represented using IDL can be stored in the interface repository. At runtime, an application can discover an interface definition and, using the DII, build a remote object request.

CORBA Object LifeCycle

CORBA provides a lifecycle service that assist with the creation and destruction, as well as other aspects of the object life cycle. For complete information, refer to www.omg.org. From this papers perspective, we will only focus on the basic functionality available for object creation and the lifecycle without using the service.

CORBA objects do not use reference counting for garbage collection. Once a program has access to a server implementation object reference, calls may be made to that object at any pint in the future. The CORBA object lifecycle attempts to mimic real-life behavior.

Using the lifecycle service, the client code can specify that it wishes to delete the server instance. The server program provides an interface for the server-side control functionality, such as resetting state or object destruction. Another common approach is to allow the server to time out and gracefully terminate. It is the responsibility of the programmer to implement objects correctly based on these principles. If the object is required to maintain state information between invocations, then the object should implement a persistence mechanism.

The fact that a server may terminate while a client still holds a reference illustrates the basic operation of the CORBA object lifecycle. A CORBA object, once created, should be available while an object reference is available. Theoretically, once an application has an object reference, it can use the object at any pint in the future. It can even pass the reference out of its own

address space, possibly to a difference machine for use by another process, as shown in Figure XXX.

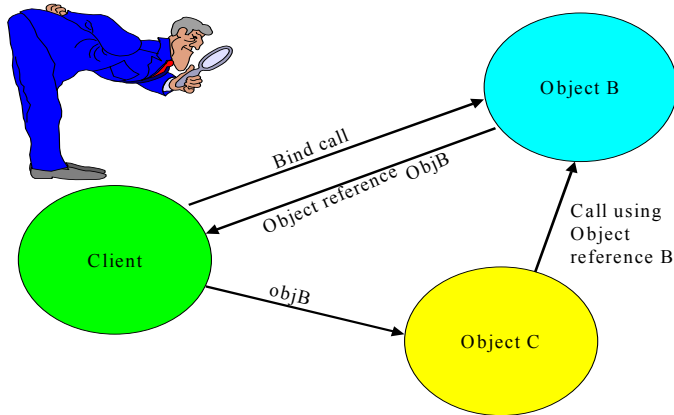


Figure 3.4 - Creating an Object Reference

The client in the diagram creates an object B. On creation of object B, the object reference for this particular instance is returned to the client. The client then passes the object reference for B across a network boundary to object C. Object C now has access to the object B.

The key to this flexibility is the object reference. The object reference contains information relating to the server that generated the reference. For example, the reference could contain the hostname and the port number of a server activation component. On receipt of a request using this object reference, the object could activate the component, taking the necessary steps to ensure that state was correctly restored.

4. Roma COM/CORBA Adapter Specifications

Candle has partnered with Visual Edge Software for the development of the Roma COM/CORBA Object Bridge. Visual Edge will be utilizing their Madrid Software for this effort. The Roma COM/CORBA Bridge has been adapted for Roma and is to be marketed as such, The Roma COM/CORBA Bridge.

Section 4 was abstracted from the Visual Edge Software Document: *Candle Roma Adapter Specification 4_14, 5th Draft*. This information is Confidential.

Overview

Using the ObjectBridge core and the COM and CORBA adapters, a COM or CORBA application can act as a client for Roma business components, sending requests to the components and, when required by the application, receiving synchronous replies. The Roma adapter creates exposable interfaces by grouping related Roma messages of a business component, through which those calls are processed.

When used from COM or CORBA a set of Roma business service appears as a COM/CORBA interface, and each Roma request/reply message pair of a business service appears as a method on that interface. Methods can have named arguments as either input, output, or both input and output parameters. Each argument translates to a field of either the request or reply (if any) message. In particular in and inout parameters are translated to fields of the request message, and out, inout, or return parameters translate to fields of the reply message.

When a COM/CORBA client calls into Roma, the Roma adapter converts the original data into the corresponding data types of the Roma message, and posts the message to the appropriate queue. If a reply is expected, an anonymous reply queue is created to receive the response, which then gets converted back into the calling adapter's native types.

There is no support for COM or CORBA components directly receiving "unsolicited" messages (i.e. COM or CORBA components cannot act as Roma business components, receiving messages from arbitrary clients). In addition, there is no support for transactional messaging (primarily due to the limitations of current versions of CORBA ORBs and COM).

Platform Coverage

The ObjectBridge Development environment is only available on Windows NT 4.0. The ObjectBridge runtime will operate on both Solaris 2.6 and Windows NT.

High Level Architecture

ObjectBridge Architecture

ObjectBridge adapters are responsible for reading repository information from a system's repository and marshalling and unmarshalling requests/replies at runtime based on the information retrieved from the repository. When the ObjectBridge is acting as a Roma client, it will be doing so in potentially multiple processes on the same machine and in multiple threads within these processes. The exact use of threads and processes can depend on the system at the other side of the bridge (e.g. COM or CORBA). Since systems like COM and CORBA are currently essentially synchronous the ObjectBridge typically uses a large pool of threads with each thread handling one request at a time. Many of these threads may be blocked waiting for replies from the target system.

In the case of COM clients, the bridge and adapter DLLs will likely run in the application process. The ObjectBridge can also be configured to run through DCOM, using the NT system surrogate to load the DCOM adapter.

In the case of CORBA clients, Roma servers will be given a naming service entry that points to a listener process known as an ObjectBridge agent. This process is contacted by the ORB naming service when the client sends an activation request. The agent will start or reuse a secondary process called a "server process", in which the actual bridging will take place. This allows each bridging process to have its own security and configuration settings, independent of each other.

Roma Adapter Architecture

The Roma adapter will be designed to act as a Roma client. It will support both fire-and-forget and synchronous calling of business services.

The adapter will support a variety of different data types (as described in this specification). The initial implementation of the adapter will use a file-based "repository" to retrieve the definition of Roma messages. For the second phase this will be updated to directly use the Roma LDAP repository to retrieve this information.

Interface Repository Architecture

The meta data "repository" used to describe the interfaces, messages, and syntax of the messages is a series of text files in a pre-defined subdirectory. Each text file represents one interface. Within the text file are definitions for all of the data types of the methods of the interface and definitions of the fields (arguments) of the messages themselves. A request and reply message are grouped together to look like one method with both input and output parameters.

When the Roma adapter browses the repository subdirectory, if an interface type is found in the directory but is not referenced in the Roma LDAP directory (through the description field of individual business components) then it is as if that interface type does not exist. Similarly, when parsing an interface description file, if a method does not correspond to a registered Roma business component then the method is ignored.

The data types that will be supported within messages are defined as follows:

Binary Types

Supported binary types must be in native platform endianness, or in that endianness which is specifically indicated for the class it is used in, with customizable (per message) alignment either on 1 byte boundary or natural alignment. They include integer and floating point, up to 64 bits.

String Types

All character types are assumed to be in the native platform character set (ASCII). The strings can be of fixed or variable length, with a length prefix or a terminating delimiter.

Complex types

Numerical Types

- Numerical types are formatted numbers stored within any of the supported string types (i.e. numerical types are stored in ASCII).

Date Types

- Date types are stored in any of the supported string types

Structured Types

These are essentially structures or records that contain a set of named fields where each field can have a different type. Field types are any of binary, string, date, or numerical types. The layout of a structure is entirely determined by the syntax of its fields (there are no special delimiters for fields of a structure itself, the delimiters of the data types of the fields themselves act as implicit delimiters between fields). Fields are not named within the message itself; field names are for use in CORBA and COM.

Array Types

Arrays come in four variants: Fixed length, length prefixed, explicitly delimited and implicitly delimited. Each element has the same type. Unless the array is explicitly delimited, there are no special delimiters between the elements of an array itself since the delimiters of the contained data type act as implicit delimiters. The elements of an array can be any of binary, string, numerical, data, or structured types.

Local Repository Issues

For phase 1, the Roma adapter will get its information about business services from a local repository, consisting of a series of text files.

The local repository will provide the following information to the adapter:

- definition of message types as structures
- definitions of a class. Classes regroup business services for which a single message pair is defined. Each of these business services correspond to one method of the class. Within Roma itself :

- The *description* of a Business service defines in which class that business service serves as a method. The *name* of that business service maps the name of the method it represents in that class. Multiple business services can then be grouped within a class by giving them all the given's class name as description.
- If there is not a one to one correspondance between Business services with the correct class description and methods within a class, those methods/business services are ignored.

Example:

If a Business service is defined, and its description field matches a class within the flat file repository, but no method of that Business service's name is defined in the class definition itself, the business service is ignored.

If a method exist in an class definition, but no according business service exists of that name (or one exists but does not have the parent class as description field), the method is ignored.

Browsing the Roma Adapter

The Roma adapter will appear to have one topmost branch:

- Local : this is the content of the local repository developed in Phase 1

When a user wants to call into Roma from CORBA or COM, it will be necessary to pick an instance from either of the local branches and add it to a service¹ exposed to COM and/or CORBA. This is just the standard way objects are normally bridged.

Synchronous Replies

For each business service message there will be a synchronous method that a COM/CORBA client can call (presuming the business service message has a corresponding reply message). The Roma adapter will be receiving requests in multiple threads at the same time. Any of these requests may be synchronous. In order to handle these in the thread that the adapter was invoked on, the adapter will open a thread-specific handle to a named unshared client. The name of the unshared client will be defined based on a configuration editor profile setting. This allows the adapter to synchronously wait for replies in whatever thread a request was made in.

5. Roma's External Specifications

Section 5 was abstracted from the Visual Edge Software Document: *External Specifications - Candle Roma adapter (Phase 2), Last Revision July 22, 1999*. This information is Confidential, with the Intended Audience of Visual Edge Development and Candle Development only.

Overview

This document describes the externals under the responsibility of Visual Edge development regarding the phase 2 of the Roma Adapter for Candle. As mentioned in the Statement of Work, there are three items addressed by Visual Edge during the development phase:

- Migrating from the original CLS file-based repository described in the Phase 1 specifications to the LDAP-based repository Candle will be providing.
- Support for Asynchronous method calls into Roma.
- Support for COM or CORBA components to handle unsolicited messages and asynchronous replies through the ObjectBridge's internal Event Routing mechanism. This will effectively permit object components to impersonate Business Elements.

Candle is responsible for providing specification/API of the Roma v3.0 repository.

LDAP Repository

Overview

Roma users will define the various repository contents using software provided by Candle. The ObjectBridge will not write into the Repository contents itself. The ObjectBridge will read from the repository in order to export its contents to other systems. The mapping of the different Roma types into other systems will remain the same as in Phase 1.

Development Work

The development team will implement the functionality necessary to read the contents of the LDAP-based repository. This functionality will be part of the deliverable adapter's source at the end of Phase 2.

Asynchronous method (Calls into Roma)

The Roma Adapter will support asynchronous calls to Roma in Phase 2, and this functionality will be provided by splitting the call process into two separate tasks: sending information out, and receiving reply information.

Exposure

The Mapping of asynchronous methods will proceed as follows:

- If the method contains no output parameters, it is mapped into foreign system as a single, asynchronous method, just as it would in Phase 1.
- If the method contains output parameters, but is not tied to any native Roma server implementation (ie, it uses a Bridge BE), it is mapped as a single synchronous method. Since these methods are intended to provide functionality to *Roma* clients, and that these given clients are already asynchronously communicating with the bridge, it does not make sense to map the async call convention in this case.
- If the method contains output parameters, and happens to be a native Roma Server, then an additional mapping is provided to provide asynchronous functionality to the eventual clients using it as described below.

When encountering an asynchronous method within the Roma Repository, the bridge will provide two extra functions in order to support the async. calling convention:

- One Fire and Forget method containing all the input parameters originally passed in the function's signature.
- A separate function dedicated to the retrieval of the actual output parameters. This function may or may not be blocking depending on a Boolean parameter appended to the mapping.

CORBA & COM Servers (Calls from Roma)

CORBA & COM servers acting as Roma Business elements are integrated through the *event framework*. The event framework defines event routes which will permit the ObjectBridge to *route* unsolicited requests to the proper servers.

Exposure

To expose a CORBA or COM server to ROMA, the user has to go through the following steps:

- The user defines a Business Element that the server will impersonate within Roma. This Business element defines an *Interface* within the Roma repository, as specified in the Repository API specifications.
- The user selects the *Interface* and exposes the interface into the foreign system.
- The user then implements the exposed interface and creates an Object in that system.
- The user then ties each *Roma method instance* to the server instance. As this implies, the actual implementation of the foreign server may be split among multiple server objects.

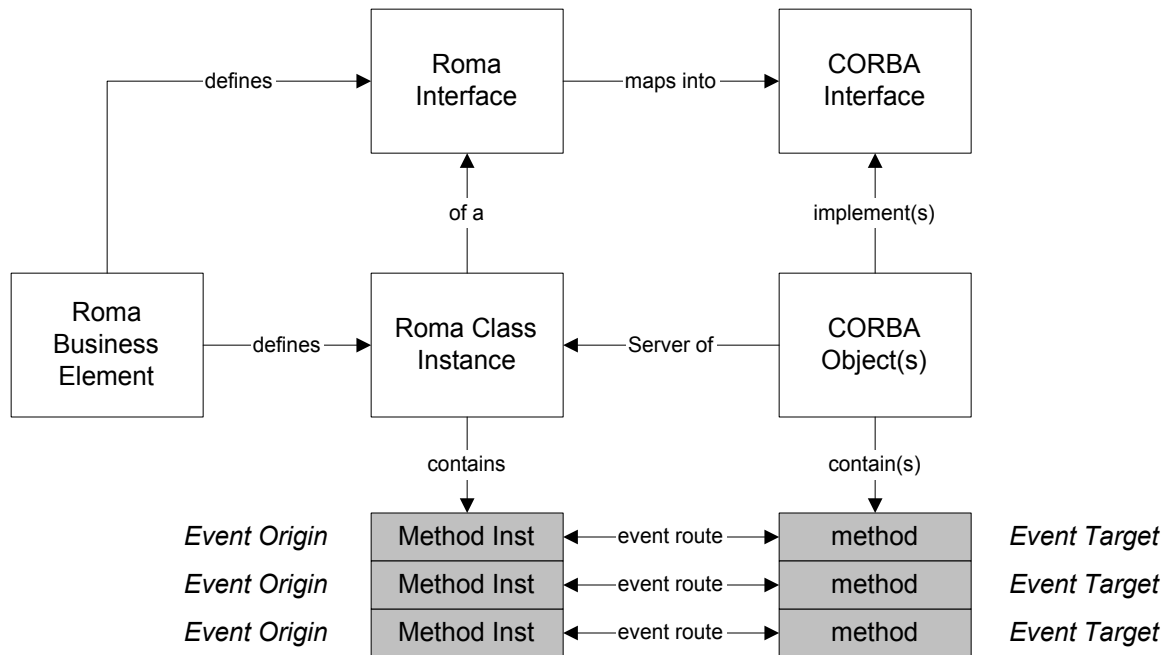


Figure 5.0 -

Exposure example

Given a certain Interface has been defined within the Roma Repository:

```

interface RomaServerIFace
{
    void fooInt(in signed32 inArg, out signed32 outArg);
    string(0) fooString (in string(0) inputString);
}
  
```

Mapping it into CORBA would look like:

```

interface CorbaServerIFace
{
    void fooInt(in long inArg, out long outArg);
    string fooString(in string inputString);
}
  
```

This interface is retrieved by exposing the original *interface* (Blue icon in the browser) into CORBA and then generating the bindings as CORBA IDL, just as one would when exposing a Native Roma Server into the CORBA system.

The implementation phase of the interface is fairly straightforward, and follows the typical CORBA conventions: more specifically, one derives the generated *skeleton* class and implements the given abstract interface originally represented by the exposed interface.

Once the server has been implemented and registered with the CORBA Interface repository, the user needs only to define the event routes linking Roma “method calls” into the newly implemented CORBA Server methods.

Technical Overview

In order to be able to service unsolicited calls into CORBA/COM servers, the development teams needs to implement an *agent* dedicated to servicing a Roma queue. This *Roma agent* uses the adapter (as the figure below shows): it is responsible for servicing incoming requests from Roma, *activating* instances of object servers, and passing the related input/reply information across. The listener agent will make use of the Roma adapter and the core’s event framework functionality to correctly dispatch the requests to their assigned servers. The figure below gives an example of an incoming request for a CORBA Server, but the same listener agent may also dispatch to the variety of systems currently supported by the ObjectBridge

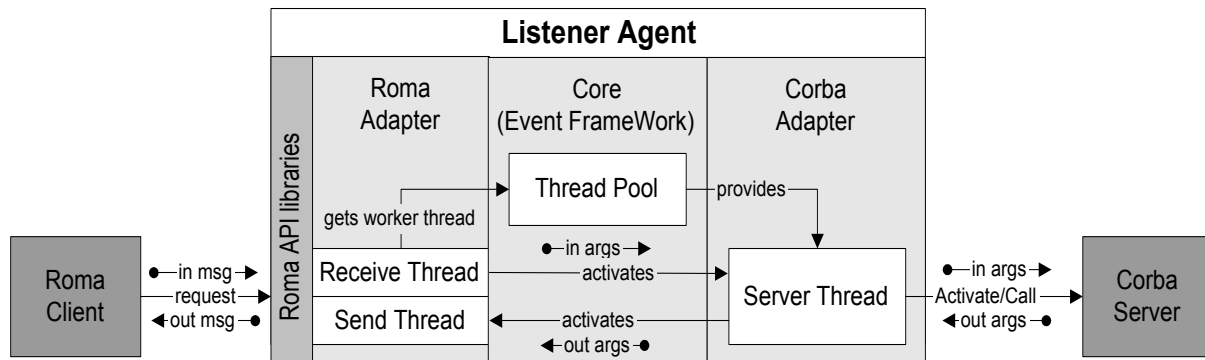


Figure 5.1 -

The Roma listener agent acts as a Business Element in the Roma world. Multiple Roma agents may be launched provided they listen on different Business Elements on different queues. The user will be able to specify a profile on the command line for the agent to use. This profile (part of the Object Bridge “Profile Editor”) will be used in order to specify which Business Element the agent will serve in the Roma environment. Multiple instances of a listener agent are useful when one wishes to simultaneously service different sets of object servers for which different profiles, configuration details, and/or application specific issues exist.

| | |
|--|---|
| VisiBroker for NT VisiBroker for UNIX | VisiBroker for C++ Version 3.1, 3.2, and 3.3.1 VisiBroker for Java Version 3.2 VisiBroker for C++ NamingService—Version 3.0.1 JDK 1.1 |
|--|---|

*Support for these object systems is not bundled with *Roma Object Access*, but can be licensed separately.

The product also supports the generic IOP protocol, which makes it usable with other IOP-based Component Support not listed above, as long as their level of compliance with the IOP standard is adequate for the generic IOP adapter included with the product.

Although equivalent concepts exist, ObjectBroker is not a fully CORBA-compliant object system and the ObjectBroker support has not been implemented along the same principles.

Messaging Software and Configuration

Roma Object Access is independent of the messaging product used by Roma.

Introduction

Purpose of This Release

The *Roma Object Access* product will be delivered in two phases.

Phase I – This deliverable. This phase provides access to Roma Business Services from the supported object system. This means that DCOM or CORBA clients can view and use Roma Business Services as methods defined on one or more objects. The Object interface definitions that map methods to Roma Business services are stored in a file and are defined using the MDL (Metadata Description Language).

Phase II – The next deliverable with BSP 3.0. This phase will provide access to objects in the object system from Roma. The interface definitions will be stored in the Roma Directory, and will use data types defined in the Roma Meta Data Repository.

Those wishing to make early usage of *Roma Object Access*, should note that the following terms apply to the phase I delivery of the *Roma Object Access*:

1. The *Roma Object Access* is provided only with samples and electronic versions of early releases of documentation.
2. Full Candle support is provided for GA status products only.
3. *Roma Object Access* phase I supports only one way access from foreign system to Roma.
4. *Roma Object Access* phase I supports only synchronous access to Roma.

What is Roma Object Access?

The *Roma Object Access* (phase I delivery) product supports access from Object Oriented systems to Roma.

Features

This section describes the features in *Roma Object Access* phase I delivery.

Roma Object Access Browser

The *Roma Object Access* Browser in this release provides the following features:

- *Browsing defined Roma Objects.* *Roma Object Access* browser can provide a tree view of all Roma objects defined using the MDL files.
- *Provide target object oriented system's view of Roma object interface.* *Roma Object Access* browser can present the interface signature of a Roma Object in the target system's language.
- *Installing and generating Roma object definitions in target O-O system.* *Roma Object Access* browser provides "one stop" operations to group Roma object definitions into a "service" and install the service and generate the binding for this service in the target O-O system.
- *Browsing target object oriented system's object definition.* The browser also allows user the browse the object interface definition in the target system. This is useful to help user check any useful information in target system and verify the installed service in target system.

Roma Object Access Agent

Roma object Access Agent simulates the existence of a Roma object in target O-O system. On any request to this simulated object, the Agent will route the request to mapped Roma BusinessService, convert the response from Roma to return values and return them to the client in target O-O system.

Roma Object Access LogView

Roma Object Access LogView provides major log related facilities related to the *Roma Object Access* product. (Available on NT only)

- *Graphic Log browser.* LogView provides a graphic interface for user the browse the log history.
- *Log Configuration.* LogView also provides interface to adjust the log related properties of the *Roma Object Access* (like: log level, log content etc).

Overview of Features in Roma BSP 3.0

The features in this release will be incorporated into Roma BSP 3.0. In addition to the features in this release, additional features will be supported.

Additional features in *Roma Object Access* for Roma BSP 3.0 (Phase II)

The Roma Object will have following additional features:

- Define the object interfaces and Roma mappings in Roma MDR (Meta Data Repository).
- Access foreign O-O system through *Roma Object Access*
- Asynchronous access
- Use ROM (Roma Object Manager) GUI interface to create and maintain Roma Object definitions.