SATHYABAMA UNIVERSITY

(Established under Section 3, UGC Act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SCSX1020 COMPONENT BASED TECHNOLOGY

SCSX1020 COMPONENT BASED TECHNOLOGY

UNIT I: COMPONENT CONCEPTS 10 hrs.

Basic components – Software components – COM/DCOM – Java beans – Enterprise java beans – CORBA – Distributed object – Request and response – Remote reference – IDL interface – Proxy – Marshalling.

Unit- I

COMPONENT CONCEPT

1. Basic Components

1.1 Software Components

The software component is a system element offering a predefined serviceable to communicate with other components. The component is a software element that conforms to a component model, and can be independently deployed and composed without modification according to a composition standard.

- Component is an executable unit of functionality.
- Self contained unit of software code consisting of own data and logic with well defined connection or interfaces exposed for communication.
- Repeated use in application with or without customization. (Reusable component)One can buy or download it, deploy it, and it works.
- Properties: - Can be deployed independently and Unit of third-party composition.

The Characteristics of an object which includes

- Unit of instantiation
- It has a unique identity
- Unlike a component, may have state, and this can be externally
- Encapsulates its state and behavior

Key:

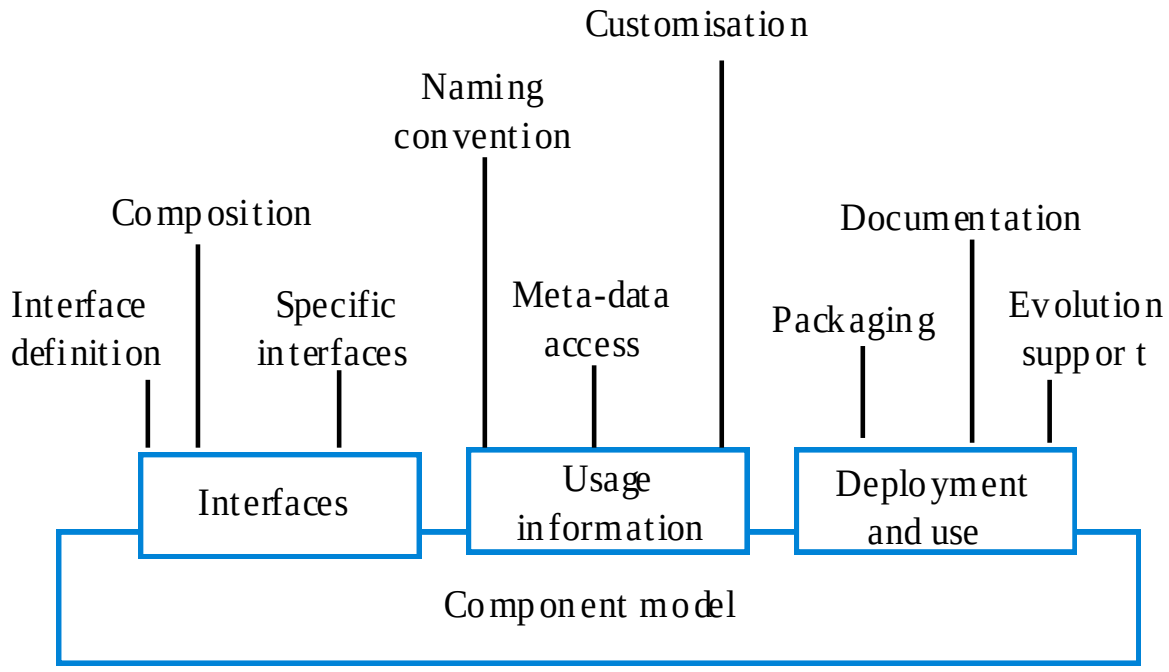
- Binary bundle of code that is compiled, linked and ready-to-use (deployable entities)
- Component implements one or more interfaces that are imposed upon
- Implementation is hidden (language-independent)
- Contract - Component satisfies obligation

- (Developed components obey certain rules in predictable ways and deployed into standard build-time and run-time environment)
- Components are stored and managed with container
- Components are standardized
- Independent deployment(eg., executable, shared library, class files etc.,)
- Interaction Standard (well defined mechanism for composition of components)
- Interfaces (Allows clients for a component access its services)
- Properties (Make it reusable and replaceable)

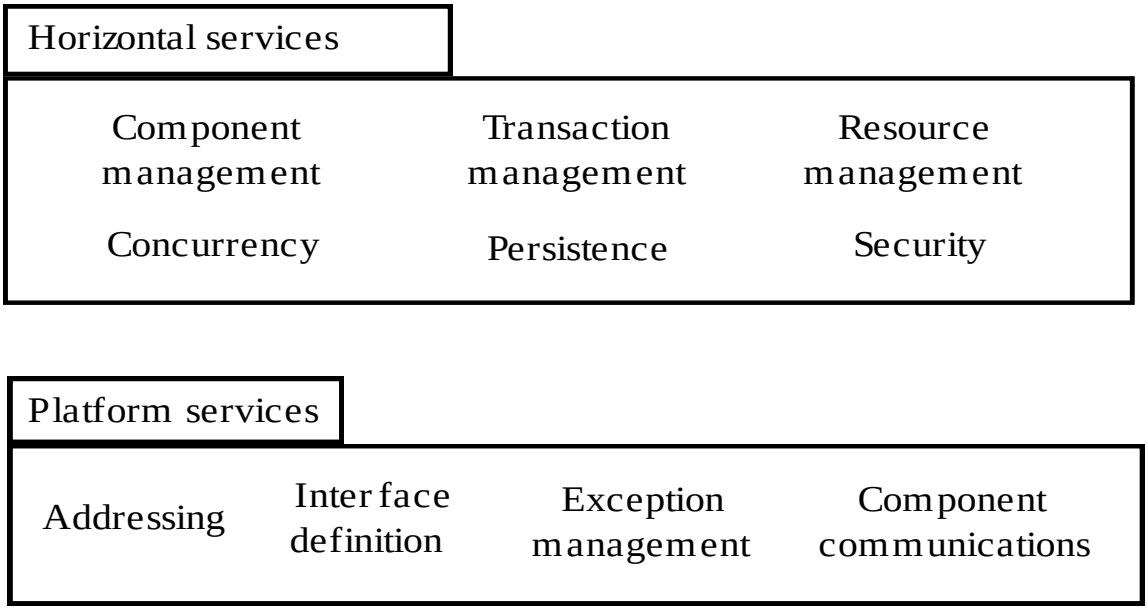
- Component framework is a dedicated and focused architecture usually around a few key mechanisms and a fixed set of policies for mechanisms at the component level.
- component model is a definition of standards for component implementation, documentation and deployment
- Specifies how interfaces should be defined and the elements that should be included in an interface definition

Examples: EJB model (Enterprise Java Beans), COM+ model (.NET model), Corba Component Model etc.,

Elements of a component model



Component model services



Interface

Interfaces allow the clients of a component to access the services provided by a component. Different interfaces will normally provide access to different services. Each interface specification could be viewed as a contract between the component and a client

Types of interfaces

(i) Procedural interfaces or Direct Interfaces

- provided directly by the component corresponding to interface of traditional libraries
- definition and implementation belong to one component

(ii) Object interfaces or Indirect Interfaces

- Provided by objects implemented by a component, corresponding to object interfaces
- Definition and Implementation might sit in different components

Properties of Interfaces

a) Interfaces define a component's Access Points

- A component can have multiple access points(interfaces) each of which provides different service.

b) Interfaces aid in specifying interaction between the components

c) Component Technology enforces a strict separation of the interfaces from its implementation

- In component Technology, providers and clients are ignorant of each other. A component

Specification names all the interfaces that a component should adhere to and adds component specific properties.

d) Interface specification bind components

- As components are bound dynamically, interface specifications are essential to glue components together to form an application

Interface serves as contract

- A contract has two sides – the client and the provider. Contracts are established by specifying the pre- and post-conditions for the operations.
- The client has to establish a pre-condition before calling the operation. The provider ensures that the pre-condition is met when the operation is called.
- The provider establishes the post-condition before returning to the client.

Differentiate objects and components

- Components are like black boxes and objects are characterised by white box reuse.
- A component can be viewed as a collection of one or more classes. The principal difference is that components are totally encapsulated.
- Components often use persistent storage whereas objects have local state.
- Components have a more extensive set of intercommunication mechanisms than objects which usually use the messaging mechanism
- Components can also be non-object oriented and can contain traditional procedures, static variables and functional or assembly programs
- Objects do not have late composition. They do not support plug and play by third party

Component Characteristics

- Standardised – Following a standard component model
- Independent – Usable without Adapators
- Composable – External interactions use public interface
- Deployable – Stand-alone entity
- Documented – Full Documentation

Benefits of software components

- Fast application development
- Reduced cost
- Software reuse
- Improved maintainability
- Ability to reconfigure software on the fly
- Ability to fine-tune the performance characteristics of real-time applications

Advantages of a component

- Management of Complexity

- Reduce Development Time
- Increased Productivity
- Improved Quality

1.2 COM/DCOM

1.2.1 COM (Component Object Model)

History of COM

- One of the first methods of interprocess communication in Windows was Dynamic Data Exchange (DDE), first introduced in 1987, that allowed sending and receiving messages in so-called "conversations" between applications
- Anthony Williams – outlined in two (internal) MS papers:
    - Object Architecture: Dealing with the Unknown or Type Safety in a Dynamically Extensible Class (1988)
    - On Inheritance: What It Means and How To Use it (1990)
- Object Linking and Embedding(OLE), Microsoft's first object-based framework, was built on top of DDE and designed specifically for compound documents. It was introduced with Word for Windows and Excel in 1991, and was later included with Windows, starting with version 3.1 in 1992
- First public version of COM shipped in OLE 2.0 (1993)
- DCOM (Distributed COM) was released in 1996
- COM+ was released along with Windows 2000 and was primarily concerned with MTS

Introducing COM

- Coming from OLE (Object linking and embedding)
- Microsoft’s Component Object Model (COM) was the first component model.
- Distributed COM (DCOM) allows creating and accessing COM objects on another machine (1996 with NT 4).
- Allows component communication
- COM components written to meet all requirements for a component architecture
- COM components consist of executable code distributed either as Win 32
- Dynamic link libraries (DLLs) or
- Executables(EXEs)
    - COM Component are fully language independent (developed and modified in any language)
    - Can be shipped in binary form
    - Can be upgraded without breaking old clients(different versions of components)

- Can be transparently relocated on a network(component on a remote system is treated the same as a component on the local system)

Software Architecture

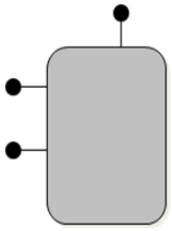
- Defines a binary standard for component interoperability
- Programming language independent
- Higher level software services like OLE
- COM supports language neutral interface called as IDL

(client application depends upon the IDL definition rather than on implementation or language details)

- Extensible by developers in a consistent manner
- Allows for shared memory management between components
- Allows dynamic loading and unloading of components
- Provides rich error and status reporting

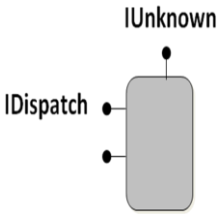
Simple COM Object

- A COM object has methods and state
  - Language independent
  - All Access is through the interface
  - Supports multiple interfaces



Specifying Interfaces

COM object appears in memory much like a C++ object. COM defines a binary standard for interfaces. Language independent implementation of the interfaces. Interfaces are defined in Interface Definition Language (IDL). An interface has an GUID as an identifier.



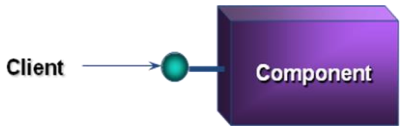
COM Architecture : A scalable programming model
COM Servers

Supports three types of servers to implement components

- In – Process server
- Local server
- Remote server

In- Process server…

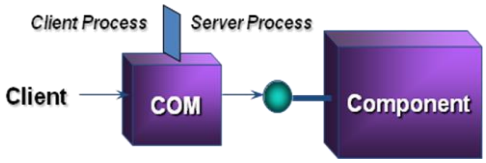
- In the same process
  - Fast, direct function calls



- Implemented as DLL
- Executes with same process space as the application
- Performance overhead of invoking the server is less

Local Server

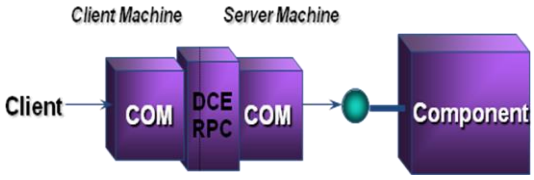
Processing is on the same machine. Fast, secure IPC



- Executes as a separate process on the same computer
- COM runtime allows communication between an application and the server (uses high speed interprocess communication protocol)

Remote Server…

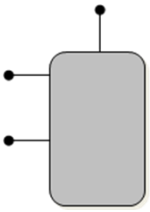
- Across machines
  - Secure, reliable and flexible
    DCE-RPC based DCOM protocol



- Execute on a remote computer
- DCOM extends COM and provides RPC based infrastructure for communication

Availability of COM

- Windows and Macintosh
- Main COM market is focused on Windows



Disadvantage

- To user COM with Java requires Microsoft JVM
- But JVM is not supported by other platforms

Interfaces and Assembly

- A COM interface is seen as a C++ virtual class and takes the form is a list of data and function declarations without associated code.
- All interfaces are descendants of the IUnknown interface.

Implementation

- A COM object is a piece of binary code, written in any programming language, as long as the compiler generates code following the binary interoperability convention.

1.2.2.DCOM

History

- DDE → OLE1 → COM → OLE → DCOM
- Dynamic Data Exchange (DDE)
    - For data exchange between any application through clipboard package
    - Originally for Windows 2.1
    - Object Linking and Embedding (OLE v1.0)
    - A compound document can *embed* objects belonging to other applications
    - E.g., an Excel spreadsheet in a Word document
    - An *embedded object* is linked to its original application
    - Restricted to document objects

Component Object Model (COM) is an Interoperability of components. It has an Ability to share non-document based components. It is based on Object-based technology

- Identity, polymorphism (multiple interfaces to a component), interface inheritance
- OLE
    - Layered on top of COM (and DCOM)
    - Links the application layer to the underlying COM architecture

DCOM implements system for Combining components those are located on different machines. DCOM protocol enables software components to communicate directly over a network. It can be Called as COM with a long wire. Distributed Component Object Model (DCOM) was designed by Microsoft. It is based on Component Object Model (COM). DCOM addresses issues such as:

- Interoperability which defines different applications, platforms, languages.
- Versioning which defiens compatibility between a new version of a server and old versions of clients.
- New interfaces should preserve the old interface
- Naming
    - Use Globally unique identifiers
- DCOM is Microsoft product, an open standard and has been ported to other platforms (on several UNIX platforms)
- Windows NT 4.0 is the first Microsoft OS to support DCOM (developed by Microsoft previously called Network OLE)
- Designed for use across multiple network transports including Internet protocols usch as HTTP

DCOM Properties

- Distributed shared memory management

- DCOM provides interfaces for distributed components to share memory
- Network interoperability and transparency
- Dynamic loading and unloading
  - DCOM manages reference counts to objects
  - Unloads objects whose reference count is 0
- Status reporting
  - Of remote execution using HRESULT struct

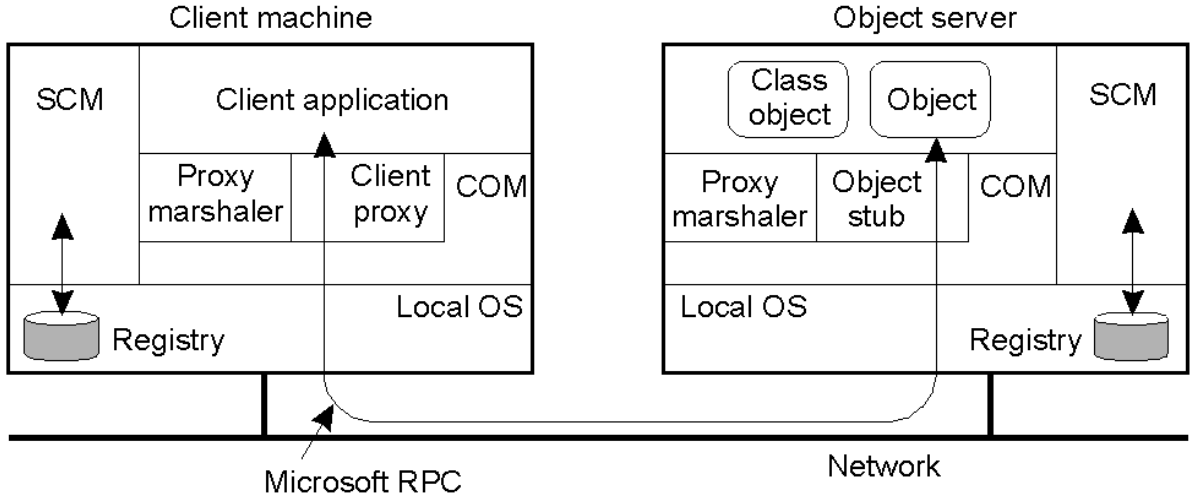
DCOM Services

- DCOM is responsible for initializing a connection between components, and
  - Negotiating protocols for communication
- DCOM provides support for persistent storage
  - Objects can persist
- Components can be assigned “intelligent names” called monikers

DCOM Model.

The Client side infrastructure in DCOM is called a proxy. The server side is called as stub. For comparing with RMI, Client side infrastructure called as stub.

DCOM Architecture



- SCM: Service Control Manager

Creating objects

- Classes of objects have globally unique identifiers (GUIDs)
  - 128 bit numbers

- Also called class ids (CLSID)

- DCOM provides functions to create objects given a server name and a class id

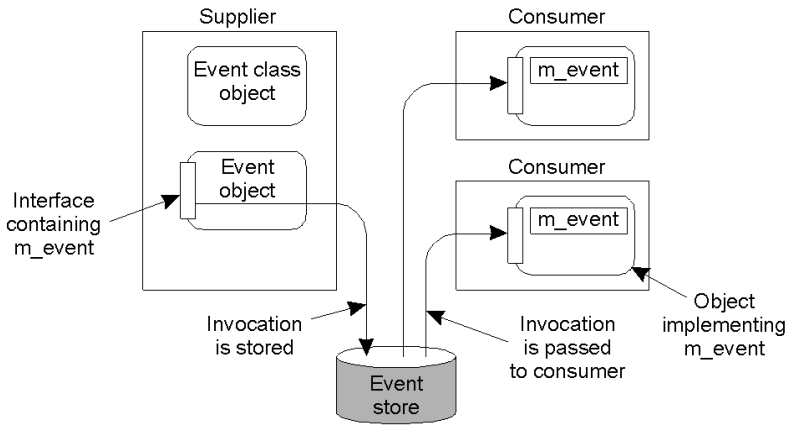
The SCM on the client connects to the SCM of the server and requests creation of the object

MIDL

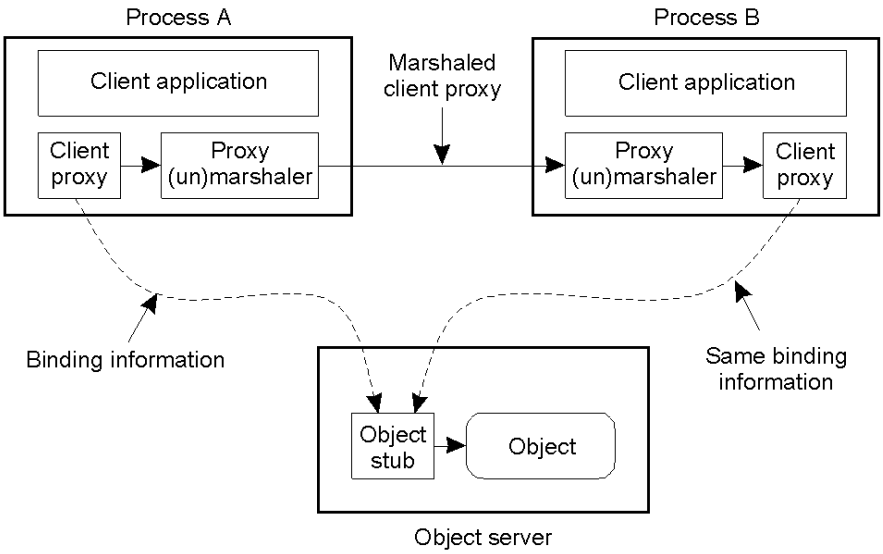
MIDL is an extension of DCE's IDL. The MIDL compiler generates the client and server stub files. Every DCOM interface inherits from an interface known as IUnknown .

- Interface names start with I
- IUnknown has three methods
  - AddRef(), Release() and QueryInterface()
  - AddRef() and Release() are used to manage reference counts (for memory management)

Events



Passing an Object Reference in DCOM (with custom marshaling)



Monikers

- Object names (as opposed to class names) are called monikers
- A moniker distinguishes one instance from another of the same class
- Monikers themselves are objects
- A moniker carries enough information to locate the object it represents
  - They can also recreate the object, if it is not currently running
- They have a human readable form similar to a URL. Example: Moniker for a file object "file:c:\my documents\July Report.doc"
- When a client passes a moniker to access an object, COM looks up a Running Object Table (ROT) for the moniker name
  - If it exists, a pointer to the object is returned
  - Else, a new object instance is created, its state is restored, its reference is entered in ROT, and a pointer to the object is returned to the client
    - Monikers contain reference to the object's persisted state

Fault Tolerance

- Supported by mean of transactions.
- Developer specify that a series of method invocations should be grouped in a transaction.

Attribute value	Description
REQUIRES_NEW	A new transaction is always started at each invocation
REQUIRED	A new transaction is started if not already done so
SUPPORTED	Join a transaction only if caller is already part of one
NOT_SUPPORTED	Never join a transaction
DISABLED	Never join a transaction, even if told to do so

Declarative Security

Authentication levels in DCOM

Authentication level	Description
NONE	No authentication is required
CONNECT	Authenticate client when first connected to server
CALL	Authenticate client at each invocation
PACKET	Authenticate all data packets
PACKET_INTEGRITY	Authenticate data packets and do integrity check
PACKET_PRIVACY	Authenticate, integrity-check, and encrypt data packets

Impersonation levels in DCOM

Impersonation level	Description
ANONYMOUS	The client is completely anonymous to the server
IDENTIFY	The server knows the client and can do access control checks
IMPERSONATE	The server can invoke local objects on behalf of the client
DELEGATE	The server can invoke remote objects on behalf of the client

Programmatic Security

- Allow applications to security levels, and choose between different security services
- Default authentication services supported in DCOM:

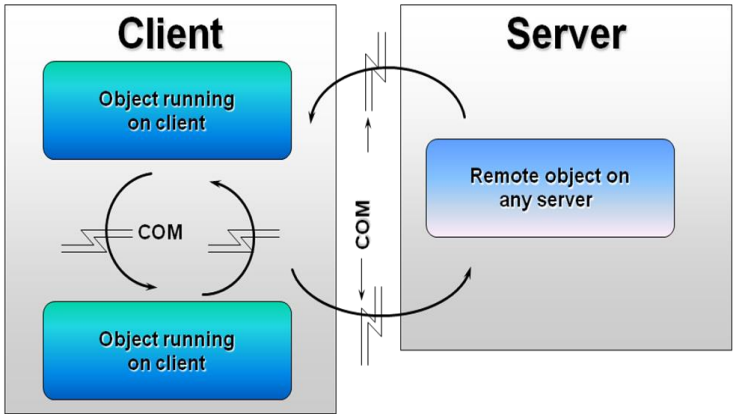
Service	Description
NONE	No authentication
DCE_PRIVATE	DCE authentication based on shared keys
DCE_PUBLIC	DEC authentication based on public keys
WINNT	Windows NT security
GSS_KERBEROS	Kerberos authentication

Default authorization services supported in DCOM

Service	Description
NONE	No authorization
NAME	Authorization based on the client's identity
DCE	Authorization using DEC Privilege Attribute Certificates (PACs)

COM/DCOM

Lets ActiveX components run anywhere



1.3 Java Beans

Introduction

Java Bean is a reusable software component that can be manipulated visually in a builder tool. JavaBeans API makes it possible to write component software in Java. Components are self-contained, reusable software units that can be visually composed into composite components, applets, applications, and servlets using visual application builder tools.

- JavaBean components are known as ***Beans***
- Provide design-time support for visual builder tools
- Simple to develop
- Leverage existing java technologies
- JavaBeans is a portable, platform-independent component model written in the Java programming language, developed in collaboration with industry leaders.

Java beans components executed in any environment

- Reused in many different application, components , websites and application builder tools
- Execution across distributed environments
- Cross-platform
- Provide a platform neutral component architecture

Key

- Assembled (graphically) in a builder tool
- Intercomponent event registration, properties
- Customization is at development time using properties, no deployment phase and roles
- Components in both client and server side

- To start the BeanBox:
  - run.bat (Windows)
  - run.sh (Unix)

BDK - Bean Development Kit

- ToolBox contains the beans available
- BeanBox window is the form where you visually wire beans together.
- Properties sheet: displays the properties for the Bean currently selected within the BeanBox window

Elements of a JavaBean

1) Properties - Private member data contained in a JavaBean.
2) Methods - Public member functions used to
   - Manipulate bean properties.
   - Expose bean functionality.
3) Events - Used to communicate changes in bean property values or changes in its state to other beans.

Features of JavaBeans

a) Support for introspection
   - Builder tool can analyze how a bean works(via reflection)
   - Reflection API (java.lang.reflect)
   - Supports run-time Class, Method, Constructor, Field info

b) Support for customization
   - when using an application builder a user can customize the appearance and behavior of a bean

c) Support for events
   - simple communication metaphor than can be used to connect up beans
   - Beans can communicate and connect together

d) Support for properties
   - Both for customization and for programmatic use

e) Support for persistence
   - (via serialization): can save and retrieve Beans data via external stores, possibly across a network
   - java.io.serializable supports Read/Write state from/to stream
   - store the values of instance variables
   - store class version (hash for class name, fields, methods)
   - so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later

Types of Java Beans

- Control beans – Used to create graphical user interface components
- Container beans – Used to hold other java beans
- Invisible Runtime beans – Used to create components that perform a specific task in the background

JavaBeans – Development Phases

- Construction phase - Involves creation of a JavaBean and its user interface
- Build phase - Involves placing the JavaBean into the target container
- Execution Phase - Involves execution of the the container application in which the JavaBean is placed

Properties

- Simple Properties - Explains how to give your beans properties
- Bound Properties - Describes how you can implement properties that when their values changes , provide event notification to other objects.
- Constrained Properties - Describes how proposed property changes can be vetoed by other objects
- Indexed Properties - Describes multiple-value properties

Steps in creating Java Bean

First, compile the Java source code. Next, create an executable JAR file.

Here are the steps to do this:

a) Compile the source code for the Bean and generate the SimpleBean.class:

b) Create a manifest file in a text editor.

c) Create the executable JAR file.

d) Load the JAR file into the BeanBox. Locate the Bean.jar file and choose it. Notice that Bean appears at the bottom of the list of Beans in the ToolBox.

e) To drop the new Bean into the BeanBox, click on SimpleBean in the ToolBox.

f) Move the cursor to any spot within the BeanBox, then click.

Writing a Simple JavaBean

Creating a Bean

The easiest way to learn to build a Bean is to start with a basic Bean, then add a property to it. The simplest Bean must implement the java.io.Serializable interface. In fact, any Java class that implements the Serializable interface is a minimal JavaBean by default. A very simple Bean, such as one that merely displays a red rectangle, extends the AWT component java.awt.Canvas. The Bean itself needs only to define a constructor that sets the size of the rectangle and sets the background color to red.

Creating a Simple Bean

Let's begin with a simple Bean that defines a rectangle of a certain size and background color (red) in its constructor. Within this red rectangle, the Bean displays a smaller rectangle in green. In addition to the constructor, the Bean includes a simple property called color that it initially sets to green. Notice that the color property is declared private.

The Bean includes two public methods, one to get the value of the color property and the other to set the value of color. Because the color property is private, it cannot be accessed directly; it can only be accessed via these getter and setter methods. Let's take a look at the code for a **SimpleBean JavaBean:**

```
import java.awt.*;
import java.io.Serializable;

public class SimpleBean extends Canvas implements Serializable
 {
    private Color color = Color.green;

    //getter method
    public Color getColor()
            {
                    return color;
             }
  //setter method
  public void setColor(Color newColor)
            {
                    color = newColor;
                    repaint();
            }
   //override paint method
  public void paint (Graphics g)
            {
                    g.setColor(color);
                    g.fillRect(20,5,20,30);
            }

  //Constructor: sets inherited properties
  public SimpleBean()
             {
                    setSize(60,40);
                    setBackground(Color.red);
             }
}
```

Serializable Interface

First of all, a Bean must implement the Serializable interface. Objects that support this interface can save and restore their state from disk. Beans that have been customized (usually by editing their properties in builder tools) must be able to save and restore themselves on request. To implement the Serializable interface, import the java.io.Serializable package and declare that your Bean implements Serializable in the class definition. Because this Bean is also visible, it extends the AWT component java.awt.Canvas.

Color Property

The color property is a private instance variable that we initialized to green:

```
private Color color = Color.green;
```

Get and Set Methods

There must be get and set methods to access each private variable. Note that the names of the getter and setter methods follow a particular format. The method names begin with either "get" or "set" and are followed by the name of the property, with the initial letter of the property name capitalized. (It is important that this format be followed so that the Bean introspection tools work.) . Thus, get and set methods have the following format:

```
public <returntype> get<Propertyname>
public void set<Propertyname> (parameter)
```

For the SimpleBean color property, we define the following pair of methods:

```
public Color getColor() { ... }
public void setColor(Color c) { ... }
```

Testing and Editing your Bean

This Bean is almost ready to be dropped into the BeanBox, where it will appear on the palette of ToolBox components.

Dropping into the BeanBox

First, compile the Java source code. Next, create an executable JAR file. The steps to do this:

a) Compile the source code for the Bean and generate the SimpleBean.class:

```
javac SimpleBean.java
```

b) Create a manifest file in a text editor.
   - The manifest file specifies the name of the class file and indicates that it is a JavaBean. The manifest file

- becomes part of the JAR file. You can name the manifest file manifest.tmp. It contains the following two lines:

  Name: SimpleBean.class

  Java-Bean: True

(On Windows, be sure to include a carriage return at the end of the text in the manifest file.)

c) Create the executable JAR file.

Use the form of the jar command to include the manifest file along with the SimpleBean.class file (Type the command on one line):

jar cfm SimpleBean.jar manifest.tmp

SimpleBean.class

d) Load the JAR file into the BeanBox.
From the BeanBox File pull-down menu, select LoadJar.

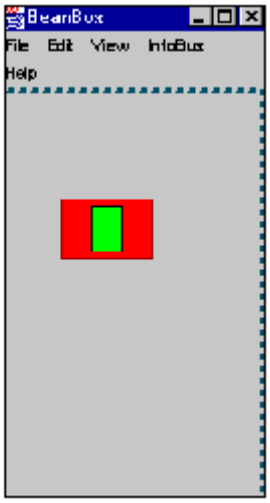
This brings up a file browser.

e) Locate the SimpleBean.jar file and choose it. Notice that SimpleBean appears at the bottom of the list of Beans in the ToolBox.

f) To drop the new Bean into the BeanBox, click on SimpleBean in the ToolBox.
The cursor changes to a cross hair.

g) Move the cursor to any spot within the BeanBox, then click.

- SimpleBean appears as a painted red rectangle with a hatched border enclosing a smaller green rectangle. The hatched border indicates that SimpleBean is selected; its properties appear in the Properties sheet.



Can reposition SimpleBean within the BeanBox by dragging on any noncorner portion of the hatched border. When repositioning a Bean, the cursor changes to crossed arrows. Because SimpleBean inherits from Canvas, you can resize it. To do so, drag a corner.

Editing the Bean's Properties

The Properties sheet displays the selected Bean's properties. SimpleBean has four properties:

- color
- background
- foreground
- name

SimpleBean declared the color property, but it inherits the other three properties from Canvas. Click on a property, such as color, to bring up a property editor in which you can change property values. The BeanBox provides default property editors for primitive types, as well as for Font and Color types.

It's easy to add other properties to the Bean.

a) Declare a new property to be a private variable.

   May initialize it to some default value when you declare it, but this is not required.

b) Declare a pair of public getter and setter methods to retrieve the property's value and set it to a new value.

   Remember to form the name of the these methods using "get" and "set" followed by the name of the variable,with the variable's initial letter capitalized.

New properties you add to the Bean appear in the Properties sheet when the Bean is opened in the BeanBox, assuming that you correctly defined the get and set methods for the property. Through the introspection mechanism, the BeanBox finds the get and set methods that match the property name and displays the property in the Property sheet.

JavaBeans and Packages

Typically, Beans (whether bought from a third party or create yourself) should be installed in a unique location relative to other Beans. Java packages help you keep everything in its proper place. Java packages are a good way to organize related classes and keep them together.

Putting Your Beans in a Package

To put your Bean in a package:

a) Add a package statement to the top of your file. The package statement must be the first line in the class file.

A package statement specifies the directory path which holds the compiled Bean class file, and the path is relative to the current working directory. Subdirectories are separated by periods **(.)** rather than slashes.

For example, for compiled Beans in the \acme\Beans directory, add the following line to the top of your file:

package acme.Beans;

The Acme03Bean, a simple JavaBean with a color property, utilizes packages. In this example you must create a directory structure so the Java compiler can put the generated class files in the proper locations. You'll also want to define the package name for classes that go in your package.

b) Import the package to files that use classes from the package.

Include the full name of the package and class when making a JAR file to include Beans that are part of a package. (JAR files are short for Java Archive files—an archive file introduced in JDK 1.1.) JAR files are the preferred packaging mechanism for shipping Beans that are built from multiple files—including both class files and image files.

c) Create the appropriate directory structure for the **acme.Beans** package below the current working directory.

First create two separate directories, one for the Java source files and the other for the Java class files:

```
mkdir -p ./src/acme/Beans
mkdir -p ./classes/acme/Beans
```

d) Copy the Java source file (or files) to the source directory in the source tree. On a UNIX system:

```
cp -p Acme03Bean.java ./src/acme/Beans
```

When you compile the JavaBean, be sure that you designate the compiled class file to go to the class directory. If you use a makefile to do your compiles, then modify the makefile accordingly.

Adding Labels to Beans

Add an instance variable to the same AcmeBean. This new variable holds a String label for the Bean. (Acme04Bean for the complete source code for this Bean.)

Adding the Label

a) Define a private variable called **label**

```
private String label;
```

b) Once the instance variable is defined, assign it a default value in the Bean's constructor. It makes sense to also set the font used to render the label. Add the following two lines to the constructor:

```
this.label="Bean"
setFont(new Font("Dialog"; Font.PLAIN, 12));
```

Eventually, this Bean will be crafted to behave like a button. You'll want to be able to customize the label for the button from within a builder tool (BeanBox, for example).

c) To enable design-time customization, make label a Bean property by adding get and set methods for it. To do this, add getLabel and setLabel methods to the Bean class.

```
public String getLabel()
{
        return label;
}
public void setLabel(String newLabel)
 {
        String oldLabel = label;

        label = newLabel;

}
```

The approach is identical to adding the **color** property.

d) While you're at it, you can render the Bean to look more like a button by adding a few lines to the end of the **paint** callback method.

```
g.fillArc(5, 5, 30, 30, 0, 360);
g.fillArc(25, 5, 30, 30, 0, 360);
g.setColor(Color.blue);
int width = size().width;
int height = size().height;
FontMetrics fm = g.getFontMetrics();
g.drawString(label,
```

```
(width - fm.stringWidth(label)) / 2,
                    (height + fm.getMaxAscent()
                     - fm.getMaxDescent()) / 2);
```

e) The definition for the paint method should look like this:

```
public void paint(Graphics g)
{
    g.setColor(BeanColor);
    g.fillRect(20, 5, 20, 30);
    g.fillArc(5, 5, 30, 30, 0, 360);
    g.fillArc(25, 5, 30, 30, 0, 360);
    g.setColor(Color.blue);
    int width = size().width;
    int height = size().height;
    FontMetrics fm = g.getFontMetrics();
    g.drawString(label, (width -
        fm.stringWidth(label)) / 2,
    (height + fm.getMaxAscent() -
        fm.getMaxDescent()) / 2);
}
```

Finishing Touches

Now when the button is drawn in the BeanBox, it looks something like a Bean. The property sheet shows fields for both the **color** and **label** properties. You can customize either one of these properties at design time by editing the property sheet. Notice that the color of the button has changed to cyan. This was done by changing the initializer for the **BeanColor** instance variable.

```
private Color BeanColor = Color.cyan;
```

This new Bean-like shape is due to the two new calls to **fillArc** in the Bean's **paint** method

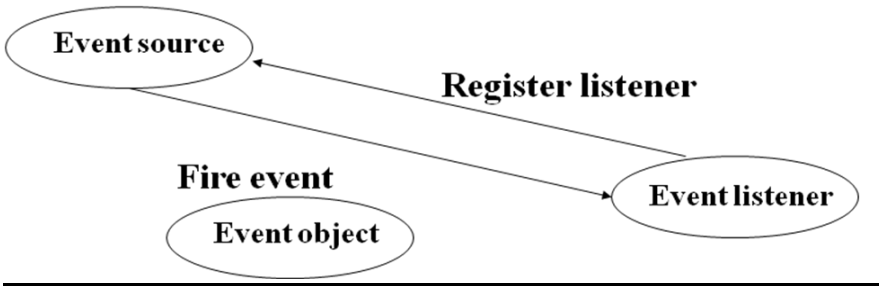
```
g.fillArc(5, 5, 30, 30, 0, 360);
g.fillArc(25, 5, 30, 30, 0, 360);
```

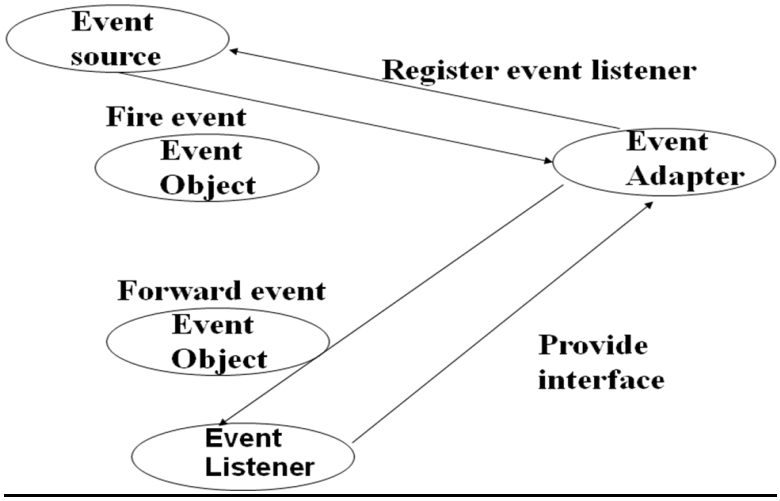
These calls add a rounded left and right side to the original square rendering of the Bean. The remaining code in the **paint** method ensures that the **String** label is centered within the button's bounding box:

```
int width = size().width;
int height = size().height;
FontMetrics fm = g.getFontMetrics();
g.drawString(label,
  (width - fm.stringWidth(label)) / 2,
            (height + fm.getMaxAscent() -
        fm.getMaxDescent()) / 2);
```

Events

- Two types of objects are involved:
    - "Source" objects.
    - "Listener" objects.
    - Based on registration
- Beans communicate via events.
- Message sent from one object to another.
- Sender *fires* event, recipient (listener) *handles* the event
- There may be many listeners
- Events and connection
    - Event Object
    - Event Adapter
    - Event Propagation
    - Event Ordering





1.4 Enterprise Java Bean

Introduction

Enterprise JavaBeans are write-once, run anywhere, middle-tier components that consists of methods that implement the business code. The enterprise bean encapsulates the business logic. Enterprise JavaBeans is the server-side component architecture for the J2EE platform. EJB enables rapid and simplified development of distributed, transactional, secure and portable Java applications.

Sun definition

"The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications that are written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification."

- EJB is a specification for creating server-side components.
- EJB Enables and simplifies the task of creating distributed objects.
- EJB components are Server-side components written using Java.
- EJB components implement the business logic only.
- EJB can maintain state information across various method calls.
- EJB components are written using Java.

Three types of Enterprise Beans

- Session bean
  - Performs a task for a client
  - Stateful vs. Stateless (Converter example was a stateless session bean)
- Entity bean
  - Represents a persisent business entity object
  - Bean managed vs. Container managed

- Message-Driven
  - Acts as a listener for the Java Message Service
  - Processes messages asynchronously

Enterprise JavaBeans Component Architecture

Consists of:

- EJB server
- EJB container
- Enterprise bean

EJB Server:

- Contains the EJB container
- EJB server provides some low level services such as network connectivity to the container.

EJB Server Provides the following services to the container:

a) Instance passivation

- If the container needs a resources it can decide to temporarily swap out a bean from the memory storage.
- This is called instance passivation. Passivation can be performed to release memory space and can also be used to synchronize the bean's state with the underlying data store.

b) Instance pooling

The EJB server enables multiple clients to share instances of enterprise beans.

For example if a client requests for a bean a new instance of the bean is created only if the instance of the bean does not exist in the memory. If an instance of the requested bean exists in the memory the bean is reassigned to another client. This is called instance pooling. This improves the efficiency as it reduces the number of instance and consequently the resources needed to service the client requests.

c) **Database connection pooling**

When an enterprise bean wants to access a database, it does not create a new database connection if the database connection already exists in the pool. Instead, it obtains a database connection from the pool. When the bean releases the database connection it can be reused by another enterprise bean.

d) Precached instances

The EJB server maintains a cache. This cache contains information about the state of the enterprise bean.

When an enterprise bean is created, it might require loading some state information. The enterprise bean

can use the precached instance to load the information. Thies increases the speed of careating an EJB.

EJB Container:

- Contains the enterprise beans
- Clients communicate with the enterprise bean through remote and home interfaces provided by the container

EJB Container Provides the following services:

a) Security

The EJB container ensures that only authorized clients can access the enterprise bean and its business methods. This is done by access control list (ACL). Access control list is a list of user groups or person authorized to access a particular functionality of the enterprise bean.

b) Transaction management

When a client calls a business method of the enterprise bean, the EJB container manages the transactions. When developing an enterprise bean specify the type of transaction support for the bean. The container proves the support according to the transaction type specified.

When the client invokes the bean's method and if the client has a transaction in progress, then the bean runs within the client's transaction Otherwise the container does not start any new transaction for the bean.

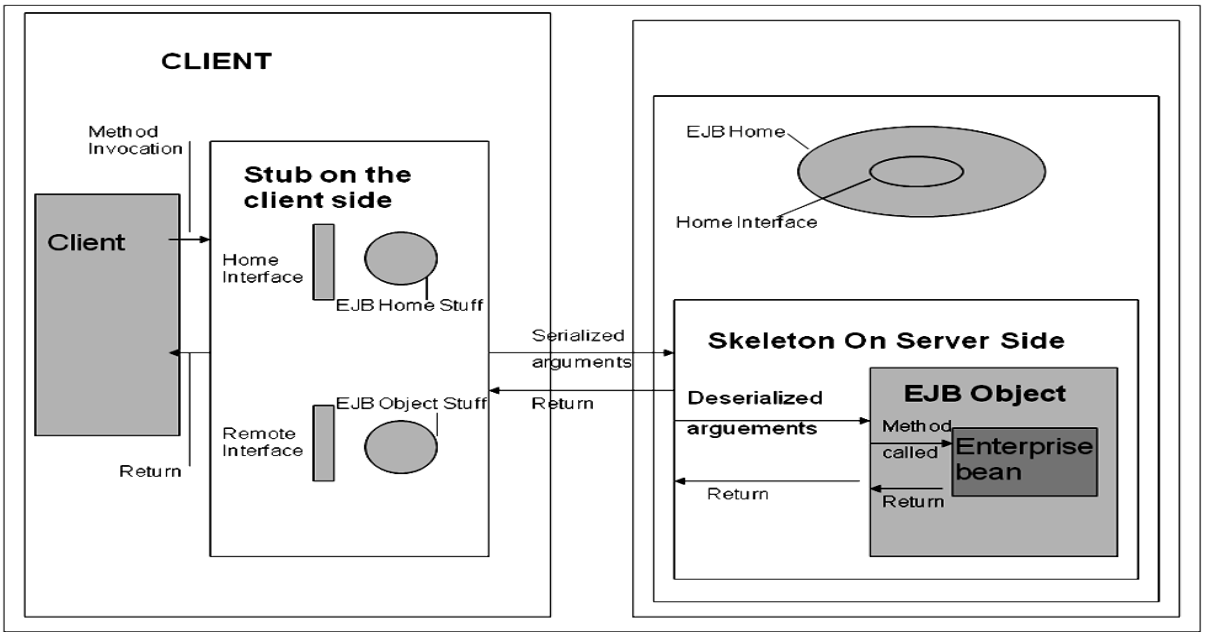
c) Persistence

The EJB container provides support for persistence. Persistence is the permanent storage of a state of an object in a data store such as database or file. This allows an object to be accessed at any time without requiring the object to be recreated every time it is required.

d) Life cycle management

The EJB container is responsible for controlling the life cycle of an enterprise bean. When a client makes a request for an enterprise bean the container dynamically instantiates, destroys and reuses the enterprise bean is required.

Communication between the client and the Enterprise bean



- The client invokes a business method on a remote interface.
- The method invocation is passed to the stub. The stub serializes the arguments into a form that can be transmitted over the network. This process is called marshalling.
- The arguments are then passed to the skeleton on the remote server. The skeleton deserializes the arguments.
- The deserialized argument is passed to the EJBObject class that implements the remote interface.
- The EJBObject class calls the business method in the enterprise bean. After the business method executes it returns a value. This value is returned to the EJBObject class.
- The EJBObject class returns it back to the skeleton.
- The skeleton serializes the return value and sends it back to the stub.
- The stub deserializes the return value and sends it to the client that invoked the business method.

Session Bean

Session bean perform business tasks without having a persistent storage mechanism such as a database and can use the shared data.

- Session EJB
  - A task or tool
  - Part of the client
  - Non-persistent

Two types of session bean

(i) Stateful session bean

(ii) Stateless session bean

- Stateless and stateful session beans, interact with their clients during method calls. This interaction between the bean and its client is known as “conversation”. The span of such conversations typically depends on the number of method calls. In stateless session beans, the conversations span only a single method –call. In stateful session beans, the conversations span multiple method – calls.
- Stateless session beans do not have instance variables to store information. Hence stateless session bean can be used in situations where information need not be used across method calls.

Stateless

- No state between methods
- All instances are equivalent
- Created and deleted by server

No conversational state maintained. Bean state (instance variables) are only relevant during a single request. All beans are equivalent. (can use any bean in pool for request).

Stateful session bean

Stateful session bean can store information in an instance variable to be used across various method calls. Some of the application require information to be stored across various method calls.

- Stateful
  - State held across methods and transactions
  - Created, used and deleted by a client
  - Lifetime bounded by timeout

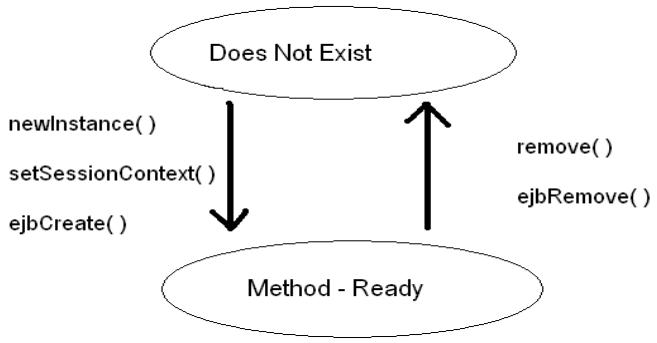
Bean's instance variables maintain state of a unique client-bean session. State is retained for duration of client-bean session. State disappears when client terminates or removes bean.

Eg.,

In a shopping site the items chosen by customer must be stored. A customer might alter the list of items selected. This list must be retained while the user browses through the shopping site. When the customer buys the items, a detailed price list might be shown to the customer. The method which calculates the amount payable might use the same information stored in the instance variable. In such situations a stateful session beans can be used.

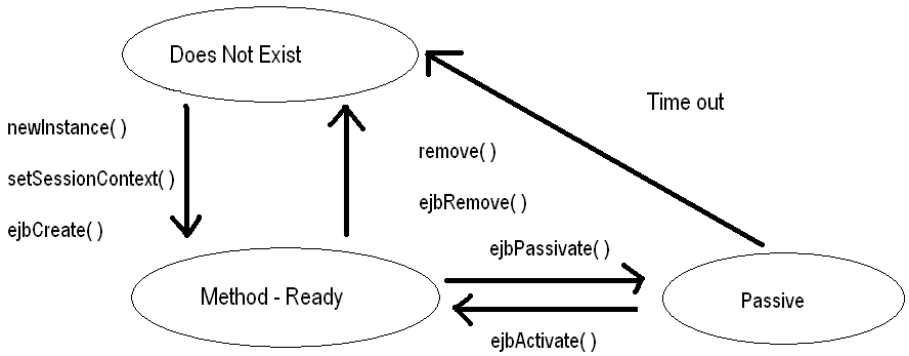
Life Cycle of a Stateless Session Bean

- The stateless session bean has two states, Does Not Exist and Method – Ready.
- Stateless session bean is in the Does Not Exist state, when the bean has not been instantiated. The enterprise bean instance enters the Method-Ready state when the container requires it.
- When stateless session bean is created, the EJB container invokes the Class.newInstance( ) method on the stateless bean class. This method creates new instance of the stateless session bean and allocates the required memory.
- The container then invokes the SesionBean.setSessionContext (SessionContext ct) method on the bean instance. This method sets the bean's reference to the session context. The session context enables the enterprise bean to interact with the customer. The enterprise bean can use the session context to query the container for information such as transactional and security state.
- The ejbCreate( ) method is finally invoked. The ejbCreate( ) method is similar to the constructor of the EJB class. It is invoked only once to in the life cycle of the stateless session bean, when the client invokes the create( ) method of the home interface. The ejbCreate( ) method must not take any argument as stateless session bean do not store any information in the instance variable.
- In Method – Ready state, the bean is ready to service the client's request. The ejbRemove( ) method ends the life cycle of the stateless session bean. This method close any open resource and frees the memory space.



Life Cycle of a Stateful Session Bean

- Stateful session bean has three states: Does Not Exists, Ready and Passive
- Stateful session bean is in the Does Not Exist state, when the bean has not been instantiated. The enterprise
- bean instance enters the Method-Ready state when the container requires it.
- Client accesses the home interface of the bean by calling the create( ) method. When the container receives this method-call, it invokes the newInstance( ) method on the bean class. The container assigns the bean instance to its EJB object.
- The container then calls the setSessionContext ( ) method on the bean instance. This method is called to set the bean's reference to the SessionContext. Then the container invokes the ejbCreate( ) method on the instance corresponding to the create( ) method invoked by the client.



- The **ejbCreate( )** method returns a reference to the EJB object's remote interface. The bean is now in the Ready state to service its client.
- When the bean is not serving any client the container can move the bean to the passive state to conserve resources. The container invokes the **ejbPassivate( )** method just before passivating the bean instance, this method is called for the bean instance to release all its resources. While passivating the bean, the instance value of the bean also known as the conversational state, are read and written to secondary storage. This is to preserve the bean's instance value when it is passivated.

- The conversational state or the bean's instance values might be primitive values serializable object or any of the following

  Home interface type (javax.ejb.EJBHome)
  Remote interface type (javax.ejb.EJBObject)
  Javax.ejb.SessionContext

During passivation the values of the fields that are declared transient are not preserved. Also it is possible that a timeout occurs for the bean instance in the passive state. The bean instance then is discarded and moved to the ***Does Not Exist*** state in such a case the **ejbRemove( )** method is not called.

- A client might make a request to the bean object that is passivated the container then activates the bean instance. Activating the bean instance involves setting the SessionContext reference variable and also restoring the bean's conversational state. After restoring the bean's conversational state the container calls the **ejbActivate( )** method on the bean instance. The **ejbActivate( )** method initializes the values of transient fields, if any and allocates necessary resources to the bean instance for servicing its client. After this method executes the bean instance is back to the Ready state, ready to service its client.
- During execution bean methods might throw system exceptions. These exceptions are not handled by the application. Some examples of system exceptions are RemoteException, CreateException and EJBException. When a system exception is thrown the container abandons the EJB object and destroys the bean instance. The bean instance thus is moved to the Does Not Exist state. In this case also the **ejbRemove( )** method is not invoked.
- When the client terminates the stateful session bean instance is also terminated. The client termination by invoking the **remove( )** method. At this stage, the container calls the **ejbRemove( )** method to terminate the bean instance.

Entity Bean

- Entity beans are enterprise that persist across multiple sessions and multiple clients
- Entity EJB
  - A business entity shared by clients
  - Persistent

Two types:

- Bean-managed persistence
- Container-managed persistence

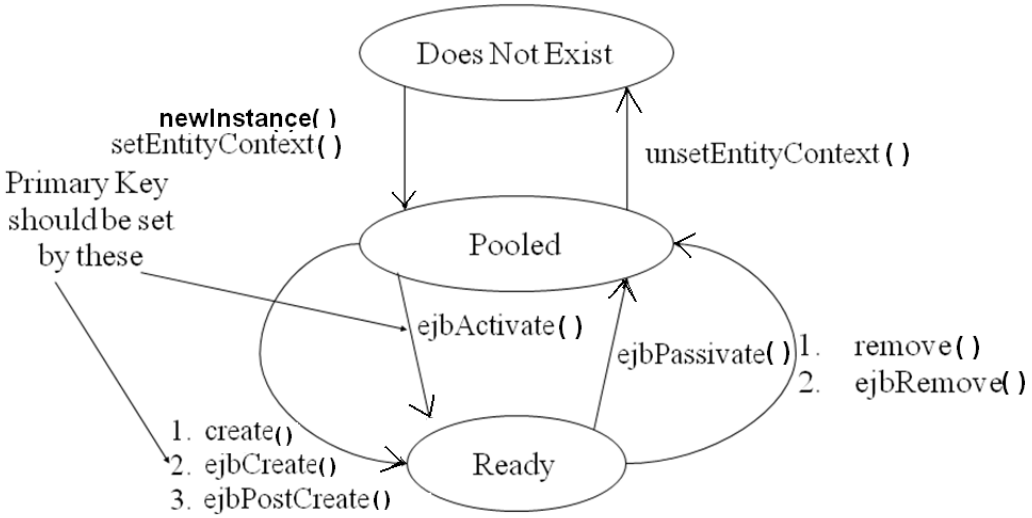
(i) Container-managed persistence - Container takes care of database calls

(ii) Bean-managed persistence - Programmer has to write the code for database calls

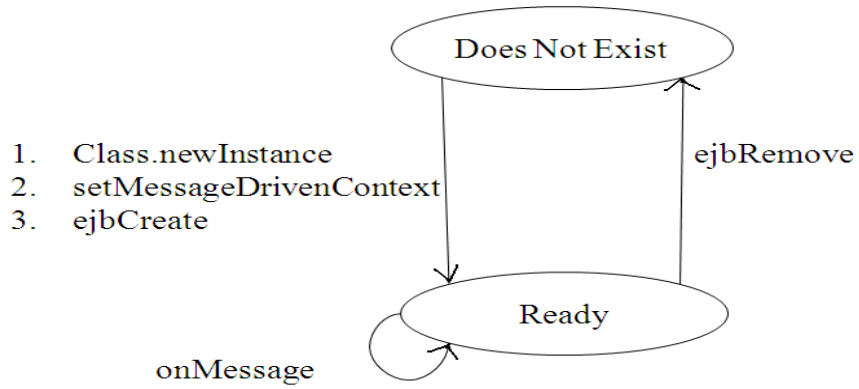
Life Cycle of a Entity Bean

- Entity bean life cycle describes the different stages in the lifetime of the bean.
- EJB container is responsible for the transition between the different stages.

- **Three different states** in the life cycle
    - Does Not Exist state
    - Pooled state
    - Ready state
- The **Does Not Exist** state represents entity beans that have not yet been instantiated by the EJB container.
- The **setEntityContext()** method is called after an entity bean has been instantiated by the EJB container.
- This method associates the bean with information about the environment such as security information.
- After instantiation the entity bean moves to the **Pooled** state. This state consists of a pool of entity beans that have been instantiated. The bean instance in the state does not have any database data or resources associated with it.
- When the client invokes the **create( )** method to access some data, the EJB container calls the **ejbCreate( )** and the **ejbPostCreate( )** methods. This causes the bean instance to move from the Pooled state to the Ready state. A bean in the Ready state is initialized with specific data (a bean is associated with a specific EJB object)
- The **ejbCreate( )** method is used to initialize an entity bean. An **ejbPostCreate( )** method is associated with every **ejbCreate( )** method. Both these methods should have the same paramenters.
- Bean instance also moves to the Ready state when the EJB container calls the **ejbActivate( )** method. This method is used to allocate resources such as sockets to the entity bean. Then invoke the business methods of the bean in the Ready state.
- When the client calls the **remove( )** method, the EJB container invokes the **ejbRemove( )** method. This causes the bean instance to move from the Ready state back to the Pooled state.
- The **ejbRemove( )** method is used to delete the data associated with the bean from the database. The bean instance also move back to the Pooled state from the Ready state when the EJB container invokes the **ejbPassivate( )** method. When the **ejbPassivate( )** method is called all the bean specific resources such as sockets are released.
- The bean instance moves from the Pooled state back to the Does Not Exist state when the EJB container calls the **unsetEntityContext( )** method. This method is used to disassociate a bean from its environment.



Message Driven Bean Life Cycle



Entity beans - Represent business object in relational database
- Entity bean ⇔ Table in relational database
- Entity bean instance ⇔Row in table

Entity beans vs session beans
- Entity beans are persistent
  - Bean managed persistence versus container managed persistence
- Allow shared access
  - Can specify transaction attributes for updates
- Have primary keys
  - Each bean has a unique identifier, used for searching
- Can participate in relationships with other entity beans
  - Primary key/foreign key type relationships are supported

Differece between Java Beans & Enterprise Java Beans

Java Beans	Enterprise Java Beans
Client or Server side	Server side
Assembled (graphically) in a builder tool	Plugged into service framework where common functions are taken care of by server (transactions, load balancing, swapping, persistence service, naming service, multitude of client protocols…)
Customization is at development time using properties, no deployment phase and roles	Customization is done at deployment time using a deployment descriptor
Inter component event registration, properties	No events or properties, beans are plugged into framework not assembled together
Java Bean component model	Enterprise Java Beans component model plus Enterprise Java Beans Framework plus Enterprise Java Beans Server specification

1.5 CORBA

CORBA Provides a Standard mechanism for defining the interfaces between components as well as some tools to facilitate the implementation of those interfaces using the developer's choice of languages

- Two features that CORBA provides are:
  - Platform Independence
  - Language Independence
- Platform Independence means that CORBA objects can be used on any platform for which there is a CORBA ORB implementation
- Language Independence means that CORBA objects and clients can be implemented in just about any programming language
- CORBA, or Common Object Request Broker Architecture, is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.
- CORBA Objects – can run on any platform , can be located anywhere on the network can be written in any language that has IDL mapping

Object Management Group (OMG) was formed in 1989. It is a consortium with a membership of more than 700 companies. It aimed at setting standards for distributed object-oriented systems, and is now focused on modeling (programs, systems and business processes) and model-based standards. Common Object Request Broker Architecture(CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms). Its aims were:

- To make better use of distributed systems.

- To use object-oriented programming (provide a common framework for developing applications using object-oriented programming techniques)
- To allow objects in different programming languages to communicate with one another
    - Frame work within which all OMG adopted technology fits two fundamental models

(i) Core Object Model
- Abstract definition
- Details how ORB facilitates distributed application development

(ii) Reference Object Model
- ORB at center, interface definitions
- Frame work for future technology adoption

Object Management Architecture (OMA)

- OMA abstracts out this common functionality from CORBA applications into a set of standard objects that perform standard, clearly-defined functions, accessed through standardized OMG IDL interfaces
- OMG standardizes the IDL interfaces and specifies what the objects do; software vendors implement and sell implementations of these objects that perform the specified services
- Applications - even if they perform totally different business tasks - share a lot of common functionality: objects notify other objects when something happens; object instances are created and destroyed and new objects' references are passed around; operation must be made secure and transactional. Beyond this, applications within a business domain (transportation, banking, whatever) share even more functionality

Applications benefit three ways from using the OMA:

- Coding is quicker, so applications can be deployed sooner: Since your crew doesn't have to write as much code, development happens faster
- Applications designed around discrete services have better architecture: Good architecture divides applications into modules or object groups based on functionality. By designing around the OMA, which is itself based on this principle, application gets a head start on its own sound architecture
- Many OMA implementations have enterprise characteristics built in: they're robust, and they scale: Providers know where you're going to deploy your applications with their OMA products, so they compete to meet enterprise's needs such as scalable name servers, transaction services, and other services

1.6 Distributed object

- The distributed objects refers to software modules that are designed to work together

- They reside either in multiple computers connected via a network or in different processes inside the same computer.
- One object sends a message to another object in a remote machine or process to perform some task.
- The results are sent back to the calling object.
- The term may also generally refer to as replicated objects or live distributed objects.

Local and distributed objects:

Life cycle :

Creation, migration and deletion of distributed objects is different from local objects.

Reference :

Remote references to distributed objects are more complex than simple pointers to memory addresses

Request Latency :

A distributed object request is orders of magnitude slower than local method invocation

Object Activation :

Distributed objects may not always be available to serve an object request at any point in time

Parallelism :

Distributed objects may be executed in parallel.

Communication :

There are different communication primitives available for distributed objects requests.

Failure :

Distributed objects have far more points of failure than typical local objects

Security : Distribution makes them vulnerable to attack.

1.7 Request-Response

Request-response or request-reply is one of the basic methods computers use to talk to each other. When using request-response, the first computer requests some data and the second computer responds to the request. Usually there is a series of such interchanges until the complete message is sent. Browsing a web page is an example of request-response communication. One can think of request-response as being like a telephone call, where you call someone and they answer the call. Compare this with one-way computer communication, which is like the push-to-talk or "barge in" feature found on some phones and two-way radios, where a message is sent without waiting for a response.

- Sending an email is an example of one-way communication.
- Request-response, also known as request-reply, is a message exchange pattern in which a requestor sends a request message to a replier system which receives and processes the request, ultimately returning a message in response.
- This is a simple, but powerful messaging pattern which allows two applications to have a two-way conversation with one another over a channel.

1.8 Remote Object Reference

A Remote Object Reference needs to uniquely identify a distributed object in the entire distributed system. To achieve this objective the Remote object reference needs to identify the server hosting the object using its IP address, the process that contains the object using the port number and unique object identifier within the process to uniquely identify the object in the process address space. A Remote object reference is created for a particular distributed object by the server process when the distributed object is created. It is needed by the client process to access the corresponding distributed object. It can be passed as parameters and as result for Remote Method Invocation. The only difference between RPC(Remote Procedure Call) and RMI is that RMI allows distributed object in the system, represented by their Remote object reference. For this purpose it requires an object adapter to translate Remote object reference to local object reference and a name service for any client to obtain Remote object reference by their names.

RMI has four layers and each layer performs a specific function.

- Application Layer: This layer contains the Client and the Server Objects.
- Proxy Layer: This layer contains the stub in the client process and the skeleton in the server process. Stub is the proxy for the server. It is used for marshalling and unmarshalling the data that is transferred to the network. Marshalling is a process which converts Java Byte codes into a stream of bytes and Unmarshalling does the reverse.
- Remote Reference Layer: It is the Object Adapter. It gets the stream of bytes from the transport layer and sends it to the proxy layer. The purpose of RRL and Object Reference is to bring about the transparent functionality of middleware.
- Transport Layer: This layer handles the actual machine to machine communication.

The process of RMI is done using the following steps. When the client process makes call to the remote function, the following steps are executed to complete the RPC.

a) The remote object which is willing to provide the service, registers in the RMI registry. The RMI register has the details of the remote object name and remote object reference.
b) The client refers the RMI registry using the object name.

c) The RMI registry returns the OR of the remote object.

d) Client procedure calls client stub using a local procedure call. Conceptually a client stub contains a set of functions with a same name a remote function. This function has the core that sends a request message to the server containing the actual parameters to the remote function, waits till it receives a reply, reads the reply and returns the result. Stubs can be generated automatically using stub compilers like RMI compiler and are included in code during compilation.

e) The client stub calls the network routines using system calls. It includes preparing the message buffer, packing parameters into buffer, building the message and issuing the sent request to the kernel.

f) In this step, the local kernel sends the message to the remote kernel. The message is copied onto the kernel and the destination address is determined then the network interface is set up and the timer is started for retransmission of the message.

g) Once the message is successfully received by the remote kernel, it is transferred to the RRL. The packet is initially checked for validity and then the stub and the RRL to which the message is to be sent are determined.

h) RRL then transfers the call to the server stub using system calls. If the stub is waiting, then the message is copied on to the server stub.

i) Once the control is handed to the server stub, it unpacks the parameters and calls the server procedure. This process is called as *unmarshalling*.

j) Server does the work and returns the result to the server stub.

k) The server stub then packs the results in a message to RRL.

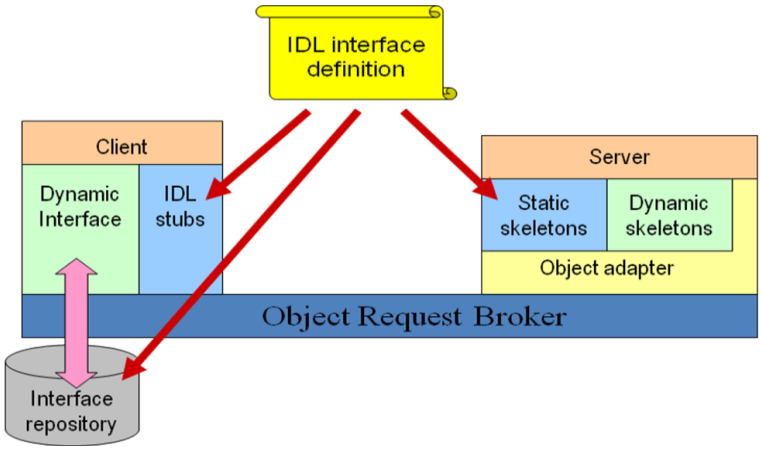
l) RRL contacts the request-reply protocol layer.

m) The server's kernel then transmits the result to the Client's kernel.

n) The Client's kernel gives the result to the client stub.

o) The Client stub unpacks the results and returns them to the client procedure.

1.9 IDL - Interface Definition Language



- Builds on OOP principle of encapsulation.
- Independent.
  - Programming Language.
  - OS
  - Platform
  - Network Connection
- Can be converted to a binary format and stored in a database

Main IDL elements

- Modules
- Interfaces
- Data types
- Constants
- Attributes
- Operations
- Exceptions

IDL data types

- Basic types
  - short long float boolean ...
- Derived types
  - using the typedef keyword
- Structured types
  - enum struct union array
- Variable types:
  - dynamic arrays, string
- The Any type

Basic types and constants

- Integer: [unsigned] short long
- Reals: float double
- 8 bits: char boolean
- Generic: any

```
const double Pi = 3.1415926 ;
const string Msg = “This is a message” ;
const unsigned long Mask =(1<<5)|(1<<7) ;
```

Structured types

```
enum CreditCard {Master, Visa, none};
struct PersonRecord {
  string name ;
  short  age ;
}
union Customer switch (CreditCard) {
  case Master:
    string cardNumber ;
  ...
}
```

Arrays, sequences, and strings

```
// arrays
typedef long longVect  [30];
typedef long longArray [2][10];
// sequences
typedef sequence <short> shortSeq;
typedef sequence <short,20> shortSeq20;
// strings
typedef string <1024> boundedString;
```

Module declaration

```
module <name>
{
  <type declarations>
  <constant declarations>
  <exception declarations>
  <interface declarations>
}
```

Interface declaration

```
interface <name> [:inheritance]
{
    <type declarations>
    <constant declarations>
```

```
    <exception declarations>
    <attribute declarations>
    <method declarations>
  }
```

Method declaration

<return type> <name> (<parameters>)
[raises <exceptions>]
[context] ;

Method parameters can be:

- in: sent to the server
- out: received from the server
- inout: both directions

Attribute declaration

attribute string name ;

readomly attribute short age ;

- Attributes:
  - are declared as variables
  - *get* and *set* methods are provided

An interface definition example

```
module Animals
{
  // Interface for a dog
  interface Dog : Animal
  {
    // a public attribute
    attribute integer age;
    // an exception that can be raised
    exception notInterested (string why);
```

1.10 Proxy Servers

The proxy server acts as a Intermediary server between clients and the actual server.

- Proxy processes request
- Proxy processes response

- Intranet proxy may restrict **all** outbound/inbound requests the intranet server

The proxy server in between client and server

- Receives the client request
- Decides if request will go on to the server
- May have cache & may respond from cache
- Acts as the client with respect to the server
- Uses one of it's own IP addresses to get page from server

Usual Uses for Proxies

- Firewalls
- Employee web use control (email etc.)
- Web content filtering (kids)
    - Black lists (sites not allowed)
    - White lists (sites allowed)
    - Keyword filtering of page content

User Perspective

- Proxy is invisible to the client
- IP address of proxy is the one used or the browser is configured to go there
- Speed up retrieval if using caching
- Can implement profiles or personalization

Main Proxy Functions

- Caching
- Firewall
- Filtering
- Logging

Web Cache Proxy

- Our concern is not with browser cache!
- Store frequently used pages at proxy rather than request the server to find or create again
- Need of web cache proxy
    - Reduce latency: faster to get from proxy & so makes the server seem more responsive
    - Reduce traffic: reduces traffic to actual server

Proxy Caches

- Proxy cache serves hundreds/thousands of users
- Corporate and intranets often use
- Most popular requests are generated only once
- Good news:

Proxy cache hit rates often hit 50%

- Bad news:

Stale content (stock quotes)

How Does a Web Cache Work?

- Set of rules in either or both
  - Proxy admin
  - HTTP header

Filtering Application Types

Proxies

- Black lists
- White lists
- Keyword profiles
- Labels

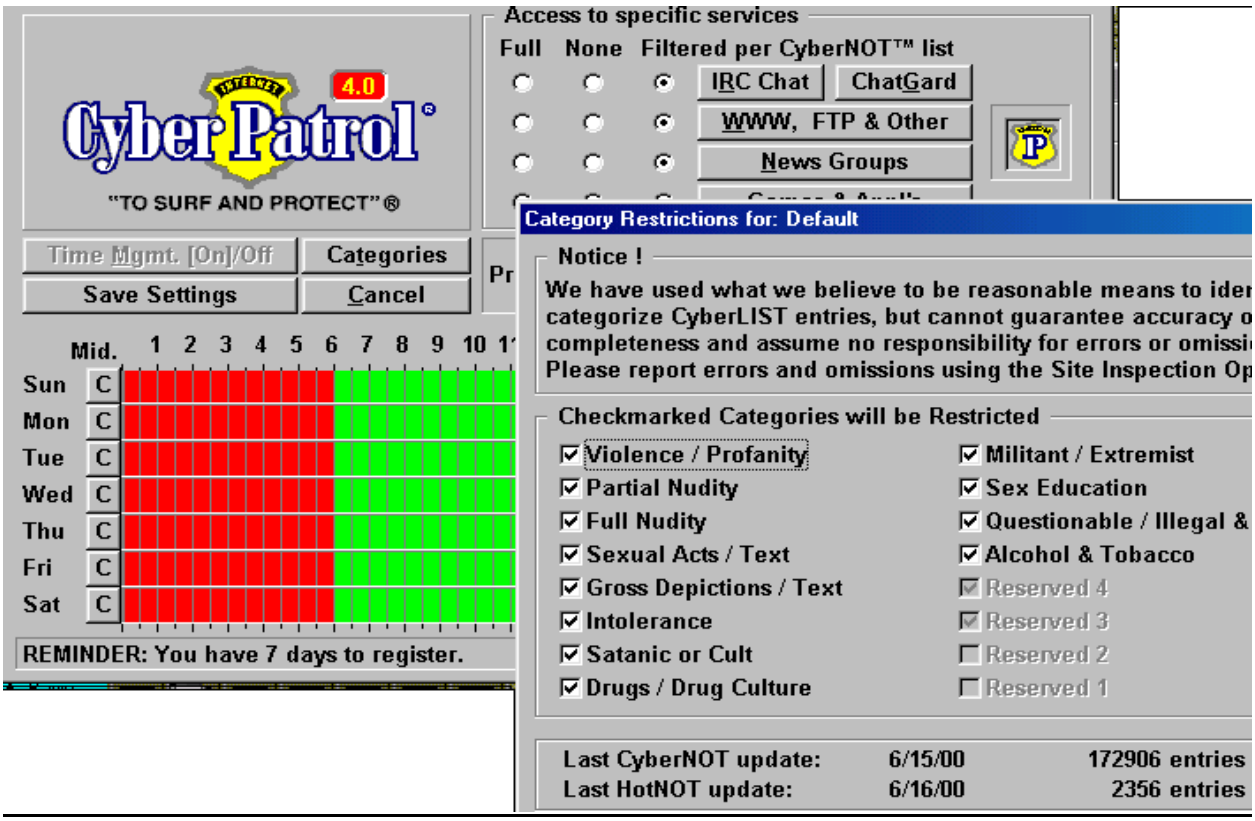
Black and White Lists

- Black list : URLs proxy will not access
- White list: URLs proxy will allow access

How Is Filtering/selection Done?

- Build a profile of preferences
- Match input against the profile using rules

Proxy level (hidden)



1.11 Marshalling

- Marshalling is the process of transforming the memory representation of an object to a data format suitable for storage or transmission.
- It is used when data must be moved between different parts of a computer program or from one program to another.
- Marshalling is a process that is used to communicate to remote objects with an object .
- It simplifies complex communication, using custom /complex objects to communicate.
- The opposite, or reverse, of marshalling is called unmarshalling

Usage

- Marshalling is used within implementations of different RPC mechanisms, where it is necessary for transporting data between processes or between threads.

UNIT – II

UNIT II BASIC CONCEPTS **10 hrs.**

Basic patterns and inherent issues – Factory – Broker – Garbage collection on client and server – Persistence of remote references – Transactions – Concurrency in server objects – Applying client/server relation recursively – Event driven programming

2.1 Basic Patterns and Inherent Issues

Design Pattern

A Pattern is a regular and intelligible form or sequence discernible in the way in which something happens or is done. A design pattern is a general reusable solution to a commonly occurring problem within a given context in the design of software. They can change the source or machine code. It is a template for solving a problem that can be used in many different situations.

Design Pattern Classification

There are three types.

(i) Creational Patterns

(ii) Structural Patterns

(iii) Behavioral Patterns

Creational Pattern

It allows to create objects rather than to instantiate it directly

Eg., creation of an object in java is through the use of "new" operator

- Factory Method - Creates an instances of several derived classes
- Abstract Factory Pattern - Creates an instances of several families of classes
- Singleton Pattern - Class of which only single instance can exist
- Builder Pattern - Separates object construction from its representation
- Prototype Pattern - Fully initialized instance to be copied or cloned

Structural Patterns

It describes how classes and objects can be combined from larger structures

- Adapter - match interfaces of different classes
- Bridge - separates an object's interface from its implementation
- Composite - tree structure of simple and composite objects
- Decorator - add responsibilities to objects dynamically

- Façade - single class that represent and entire subsystem
- Flyweight - fine grained instance used for efficient sharing
- Proxy - object representing another object

Behavioral Patterns

It passes requests between a chain of objects

- Chain of responsibility - way of passing a request between a chain of objects
- Command - Encapsulate a command request as an objects
- Interpreter - way to include language elements in a program
- Iterator - Sequentially access the elements of a collection
- Mediator - defines simplified communication between classes
- Momento - capture and restore an object's internal state
- Observer - way of notifying change to a number of classes
- State - Alter an object behavior when its state changes
- Strategy - Encapsulates an algorithm inside a class
- Template Method - Defer the exact steps of algorithm to a subclass
- Visitor - defines a new operation to a class without change

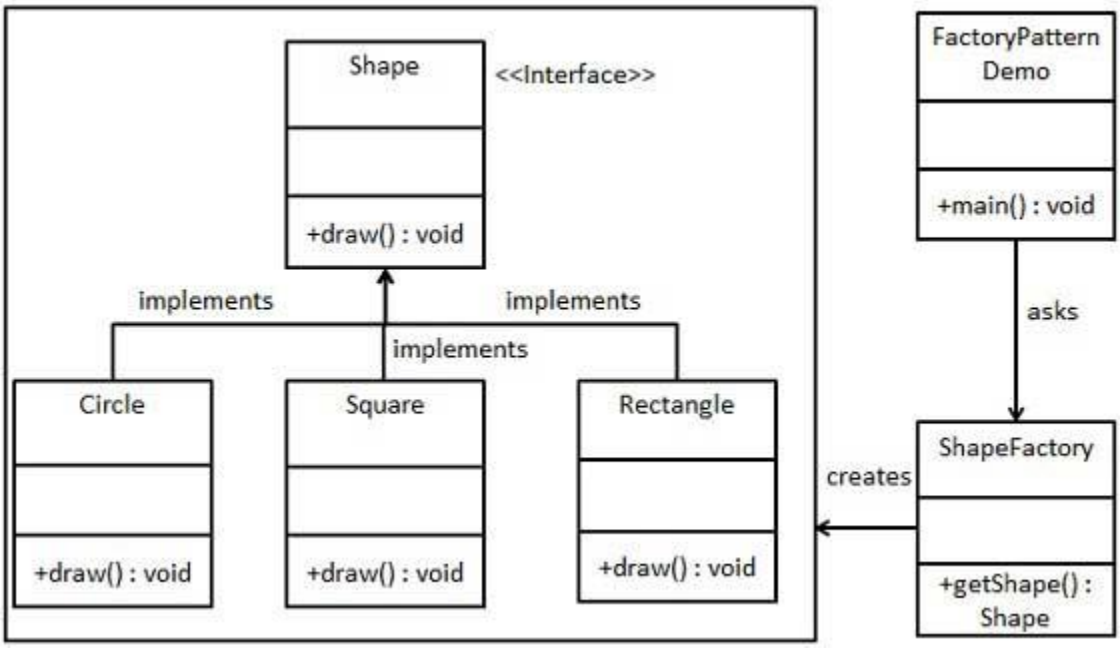
2.2 Factory pattern

Factory pattern is a kind of creational pattern provides a way to create an object.This is one of the most used design pattern in Java. In Factory pattern, creation of an object without exposing the creation logic to the client and refer to newly created object using a common interface.

Example:

This is an example of having a class named "ShapeFactory",an interface named "Shape". To create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape*object. It will pass information (*Circle / Rectangle / Square*) to*ShapeFactory* to get the type of object it needs.



Step 1

Create an interface.

Shape.java

```
Public interface Shape
{
  Void draw()
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
Public class  Rectangle implements Shape
{
 @override
Public void draw()
{
System.Out.Println("Inside Rectangle::draw() method.");
}
}
```

Square.java

```
Public class  Square implements Shape
{
 @override
Public void draw()
{
System.Out.Println("Inside Square::draw() method.");
}
}
```

Circle.java

```
Public class  Circle implements Shape
{
 @override
Public void draw()
{
System.Out.Println("Inside Circle::draw() method.");
}
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
Public class  ShapeFactory
{
 // use getshape method to get of type shape
Public shape getshape(String shapeType)
{
  If (shapeType == null)
{ return null; }
If (shapeType.equalsIgnoreCase("CIRCLE"))
{ return new Circle(); }
Else If (shapeType.equalsIgnoreCase("RECTANGLE"))
{ return new Rectangle(); }
Else If (shapeType.equalsIgnoreCase("SQUARE"))
```

```
{ return new Square(); }
return null;
}
}
```

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
Public class FactoryPatternDemo
{
   Public static void main(String [] args)
{
 ShapeFactory shapeFactory = new ShapeFactory();
//get an object of Circle and call its draw()
Shape shape1= ShapeFactory.getShape("CIRCLE");
//call draw () of Circle
Shape1.draw();
//get an object of Rectangle and call its draw()
Shape shape2= ShapeFactory.getShape("RECTANGLE");
//call draw () of Rectangle
Shape2.draw();
//get an object of Square and call its draw()
Shape shape3= ShapeFactory.getShape("SQUARE");
//call draw () of Square
Shape3.draw();
}
}
```

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

2.3 Broker Pattern

Broker Pattern is a kind of architectural Pattern which is similar to RPC but presented in a system wide scale.The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocationsThe Broker component is responsible for coordinating communication, such as forwarding requests, as well as transmitting results and exceptions.

Example

Build a City Information System (CIS) designed to run on a wide area network.Wide area network in this case can be viewed as a set of networks that are not directly connected at all time.There are different kinds of computers in the network.Users can access the information from the WWW which is the front end software supported for on-line retrieval of information. The backend are data distributed out across the network.The system will glow continuously, so individual services should be decoupled from each other.One solution is to rebuild a new network that connect everyone at all time .So, we will have to go with Internet approach. our environment is a distributed and possibly heterogeneous system with independent cooperating components.

When distributed components communicate with each other, some means of inter-process communication is required.(If component handles communication, dependencies surface such as clients need to know the location of servers).Services for adding, removing, exchanging, activating and locating components are also needed. From developer points of view, there should not be any different in doing centralized software or distributed software. We need to balance the following forces: Components should be able to access services provided by others through remote, location-transparent service invocations. Need to exchange, add or remove components at run-time. The architecture should hide system and implementation-specific details from the users of components and services.

Solution

Introduce a broker component to achieve better decoupling of clients and servers. Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker

.The broker will locate the appropriate server, forward the request to the server and transmit results and exceptions back to the client.

Using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication. This AP allows dynamic change, addition, deletion, and relocation of objects. From a developer point of view, distribution is transparent. You talk to a Broker one way or the other and it introduces an object model in which distributed services are encapsulated within objects. Broker thus supports integration of distribution as well as object technology.

Structure

There are six types of components in Broker architectural pattern: Clients, Servers, Brokers, Bridges, Client-side proxies and Server-side proxies.A server implements objects that expose their functionality through interfaces that consist of operations and attributes. These interfaces are either through Interface Definition Language (IDL), through a binary standard, or some kind of APIs. Interfaces are grouped by semantically-related functionality. So, we could have Servers offering common services to many application domains. Servers implementing specific functionality for a single application domain or task.

Class

- Server

Responsibility

- Implements services
- Registers itself with the local broker
- Sends responses and exceptions back to the client through a server-side proxy

Collaborators

- Server-side Proxy

Broker

Clients are applications that access the services of at least one server. To call remote services, clients forward requests to the broker. Clients then will receive responses or exceptions from the broker. Remember, clients and servers are logically terms. So, a client can be a server and vice

versa. Clients are the requesters and do not need to know about the location of the servers.As a developer, you can consider clients are application and servers are libraries

Class

- Client

Responsibility

- Implements user functionality
- Sends requests to servers through a client-side proxy

Collaborators

- Client-side Proxy

Broker

A broker is a messenger that is responsible for the transmission of requests from clients to servers. It also takes care of the transmission of responses and exceptions back to the client. A broker will store control information for locating the receivers (servers) of the requests (it is normally down with some unique system identifier, IP address might not be enough).Broker also offer interface for clients and servers.

They are usually in the form of APIs with control operations such as registering servers and invoking server methods. Broker keeps track of all the servers information it maintains locally. If a request comes in and it is for a server that is on the tracking list. It passes the request along directly.

If the server is currently inactive, the broker activates it (spawn a job, folk a process, create a thread, use a prestart job, etc).Then response are passed back to the client through the broker.

If the request is for a server hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route (So, a common way of communication among brokers is needed).

Sometimes, name services (DNS, LDAP, NT directory ?, RMI ?, etc) or marshaling support will be integrated into the broker.

Class

- Broker

Responsibility

- Registers (Unregister) servers
- Offers APIs (or some kind of common interface)
- Transfer messages
- Error recovery
- Interoperates with other brokers through bridges
- Locates servers

Collaborators

- Client
- Server
- Client-side Proxy
- Server-side Proxy

Bridge

Client-side proxies represent a layer between clients and the broker.This additional layer provides transparency, which a remote object appears to the client as a local one.It hides the implementation detail such as:

- Inter-process communication mechanism used for message transfer between clients and brokers
- The creation and deletion of memory blocks (remember, we are dealing with multiple different languages to build a system, not just Java)
- The marshaling of parameters and results

In most cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client

Class

- Client-side Proxy

Responsibility

- Encapsulates system-specific functionality
- Mediates between the client and the broker

Collaborators

- Client
- Broker

Server-side proxies are generally analogous to Client-side proxies

They receive requests, unpack messages, unmarshalling parameters and call the appropriate service (in a server).(Most web server has "built-in" client and server proxy to communicate with client and internet router, so there is nothing to implement in that case).(In RPC, client-side and server-side proxies are kind of parallel to stubs).

Class

- Server-side Proxy

Responsibility

- Calls services within the server
- Encapsulates system-specific functionality
- Mediates between the server and the broker

Collaborators

- Server
- Broker

Bridges are used to bridge communication among brokers

They are optional to hide the implementation detail

They are most useful when the system run across heterogeneous network (so bridge will take care of, say IPX network talks to IP network as well as SNA network)

Class

- Bridge

Responsibility

- Encapsulates network-specific functionality
- Mediates between the local broker and the bridge of a remote broker

Collaborators

- Bridge
- Broker

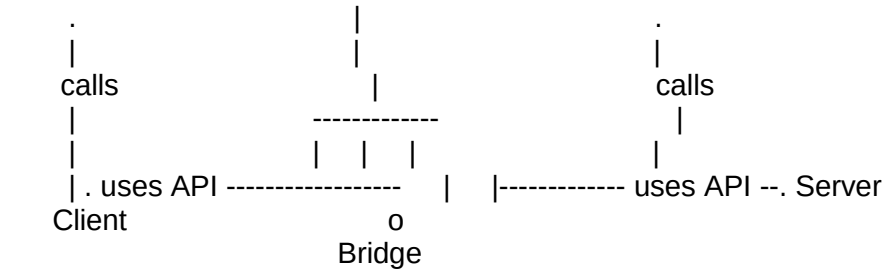
Types

There are two general types of Brokers: Direct communication and indirect communication.

Direct communication

Assume client and server understand the same protocol. Broker will assist in the initial hand share of the client and server. Then all the messages, exceptions and responses are transferred between client-side proxies and server-side proxies without using the broker as an intermediate layer.(Usually a mix approach is used, Java applet can use socket directly but not all the other components)

Brief Object Model



Dynamics

Scenario I (A server registers itself with the local broker component)

- The broker is started in the initialization phase of the system
- The broker enters its event loop and waits for incoming messages
- The user, or other components, starts a server application
- First the server executes its initialization code and then it registers itself with the broker
- The broker receives the incoming registration request from the server
- It extracts all necessary information from the message and stores it into one or more repositories (The repositories are used to local and activate servers)
- An ack is sent back
- After getting the ack from the broker, the server enters its main loop waiting for incoming client request

Scenario II (A client sends a request to a local server, local means locally maintain by the broker)

(We assume the client will block until a response is back - thus, we are doing synchronous)

- The client application starts and in the middle, it invokes a method of a remote server object
- The client-side proxy packages all parameters and other control information into a message and forwards it to the local broker
- The broker looks up the location of the server requested in the repositories (sometimes called directory).
- Since the server is available locally, the broker forwards the message to the corresponding server-side proxy
- The server-side proxy unpacks all the parameters and other control information (such as method to call, etc)
- It then invokes the appropriate service in a server
- The service execution is complete, the server returns the result to the server-side proxy, which packages it into a message with other relevant information and passes it to the broker
- The broker forwards the response to the client-side proxy
- The client-side proxy receives the response, unpacks the result and returns to the client application
- The client process continues with its computation and go on to other application logic.

Scenario III (Interaction of different brokers via bridge components)

[There are two Brokers (A & B) and two Bridges (A& B) in this scenario]

- Broker A receives an incoming request
- It locates the server responsible for the request by searching in its respositories
- The corresponding server is available but at another network node
- The broker forwards the request to a remote broker
- The message is passed from Broker A to Bridge A (This bridge will convert the message from the protocol defined by Broker A to a network-specific but common protocol understood by the two participating bridges
- After message conversion, Bridge A transmits the message to Bridge B
- Bridge B maps the incoming request from the network-specific format to a Broker B-specific format
- Broker B perfroms all the actions necessary when a request arrives as described in the first step of this scenario

2.4 Garbage Collection on Client and Server

Garbage collection

In General, Garbage collection (GC) is a way of managing the memory automatically. The *garbage collector* reclaims memory occupied by objects that are no longer in use by the program. Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC.

The basic principles of garbage collection are to find data objects in a program that cannot be accessed in the future, and to reclaim the resources used by those objects.

Usage in Programming Languages

Many programming languages require garbage collection, either as part of the language specification (for example, Java, C#, D language, Go and most scripting languages) or effectively for practical implementation (for example, formal languages like lambda calculus);

these are said to be *garbage collected languages*. Other languages were designed for use with manual memory management, but have garbage-collected implementations available (for example, C and C++). Some languages, like Ada, Modula-3, and C++/CLI, allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects; others, like D, are garbage-collected but allow the user to manually delete objects and also entirely disable garbage collection when speed is required.

Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced.

Reference counting

Reference counting is a form of garbage collection whereby each object has a count of the number of references to it. Garbage is identified by having a reference count of zero. An object's reference count is incremented when a reference to it is created, and decremented when a reference is destroyed. When the count reaches zero, the object's memory is reclaimed.

As with manual memory management, and unlike tracing garbage collection, reference counting guarantees that objects are destroyed as soon as their last reference is destroyed, and usually only accesses memory which is either in CPU caches, in objects to be freed, or directly pointed by those, and thus tends to not have significant negative side effects on CPU cache and virtual memory operation. If two or more objects refer to each other, they can create a cycle whereby neither will be collected as their mutual references never let their reference counts become zero. Some garbage collection systems using reference counting like CPython specific cycle-detecting algorithms. Reference counting requires space to be allocated for each object to store its reference count. The count may be stored adjacent to the object's memory or in a side table somewhere else, but in either case, every single reference-counted object requires additional storage for its reference count. Memory space with the size of an unsigned pointer is commonly used for this task, meaning that 32 or 64 bits of reference count storage must be allocated for each object. In naive implementations, each assignment of a reference and each reference falling out of scope often require modifications of one or more reference counters. However, in the common case, when a reference is copied from an outer scope variable into an inner scope variable, such that the lifetime of the inner variable is bounded by the lifetime of the outer one, the reference incrementing can be eliminated. The outer variable "owns" the reference. In the programming language C++, this technique is readily implemented and demonstrated with the use of const references.

Reference counting in C++ is usually implemented using "smart pointers" whose constructors, destructors and assignment operators manage the references. A smart pointer

can be passed by reference to a function, which avoids the need to copy-construct a new smart pointer (which would increase the reference count on entry into the function and decrease it on exit). Instead the function receives a reference to the smart pointer which is produced inexpensively.

Client- Server Systems

In non distributed systems, object creation and destruction automatically done by application programmers. In distributed systems, there are different approaches in dealing with garbage collection. One way is to ignore garbage collection and the operating system is killing the objects during system shutdown. The second one is done with ORB (object Request Broker) mediated object caching and load balancing in which ORB keeps track of outstanding connections. The third approach is reference count approach .Java version of COM uses first approach, CORBA ORB uses second approach and C++ Com uses the third approach.

2.5 Persistence of Remote References

Object Persistence

With object serialization Java applets and applications can save and load the state of objects to disk or over a network. One of the most critical tasks that applications have to perform is to save and restore data. Whether it be a word processing application that saves documents to disk, a utility that remembers its configuration for next time, or a game that sets aside world domination for the night, the ability to store data and later retrieve it is a vital one. Without it, software would be little more effective that the typewriter - users would have to re-type the data to make further modifications once the application exits.

Writing the code for saving data, however, can become boring repetitive work. First, the programmer must create a specification document for the proposed file structure. Next, the programmer must implement save and restore functions that convert object data to & from primitive data types, and test it with sample data. If the application later requires new data to be stored, the file specification must be modified, as well as the save and restore methods. Take it from someone who's been there - creating save & restore functions is not a fun task. The solution to this is object serialization.

Object serialization takes an object's state, and converts it to a stream of data for you. With object serialization, its an easy task to take any object, and make it persistent, without writing custom code to save object member variables. The object can be restored at a later time, and even a later location. With persistence, we can move an object from one computer to another, and have it maintain its state. This very cool feature, in Java, also happens to be very easy to use.

Serializing Objects

Java makes it easy to serialize objects. Any object whose class implements the java.io.Serializable interface can be made persistent with only a few lines of code. No extra methods need to be added to implement the interface, however - the purpose of the interface is to identify at run-time which classes can be safely serialized, and which cannot. The programmer, need only add the implements keyword to your class declaration, to identify your classes as serializable.

```
public class UserData implements
java.io.Serializable
```

Now, once a class is serializable, we can write the object to any OutputStream, such as to disk or a socket connection. To achieve this, we must first create an instance of java.io.ObjectOutputStream, and pass the constructor an existing OutputStream instance.

```
// Write to disk with FileOutputStream
FileOutputStream f_out = new
      FileOutputStream("myobject.data");

// Write object with ObjectOutputStream
ObjectOutputStream obj_out = new
      ObjectOutputStream (f_out);

// Write object out to disk
obj_out.writeObject ( myObject );
```

Note: Any Java object that implements the serializable interface can be written to an output stream this way - including those that are part of the Java API. Furthermore, any objects that are referenced by a serialized object will also be stored. This means that arrays, vectors, lists, and collections of objects can be saved in the same fashion - without the need to manually save each one. This can lead to significant time and code savings.

Restoring Objects from a Serialized State

The one catch is that at runtime, can never be completely sure what type of data to expect. A data stream containing serialized objects may contain a mixture of different object classes, so you need to explicitly cast an object to a particular class. If you've never cast an object before, the procedure is relatively straightforward. First check the object's class, using the instance of operator. Then cast to the correct class.

```
// Read from disk using FileInputStream
FileInputStream f_in = new
      FileInputStream("myobject.data");

// Read object using ObjectInputStream
ObjectInputStream obj_in =
      new ObjectInputStream (f_in);
```

```
// Read an object
Object obj = obj_in.readObject();

if (obj instanceof Vector)
{
      // Cast object to a Vector
      Vector vec = (Vector) obj;

      // Do something with vector....
}
```

Further Issues with Serialization

It relatively easy to serialize an object. Whenever new fields are added to an object, they will be saved automatically, without requiring modification to your save and restore code. However, there are some cases where this behavior is not desirable. For example, a password member variable might not be safe to transmit to third parties over a network connection, and might need to be left blank. In this case, the transient keyword can be used. The transient field indicates that a particular member variable should not be saved. Though not used often, it's an important keyword to remember.

```
public class UserSession implements
      java.io.Serializable
{
      String username;
      transient String password;
}
```

2.6 Transactions

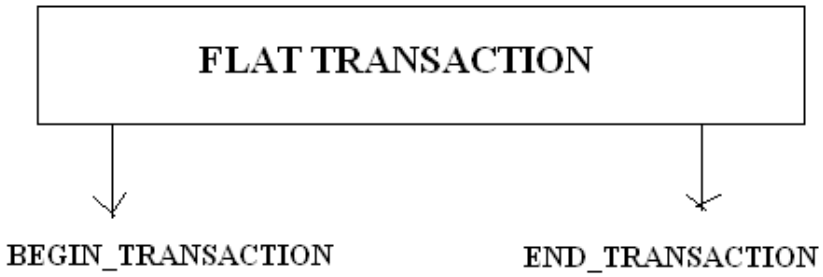
Transaction is defined as a set of actions to be performed after certain events occurs in the database. Transaction should be either performed completely or not at all. When transaction is finished, It should leave an object in consistent state. The outcome of the transaction must be persistent (no loss). It should be performed in isolation from the other concurrent transactions.

The transactions can be

i. Flat transaction,
ii. Chained transaction or Nested transaction

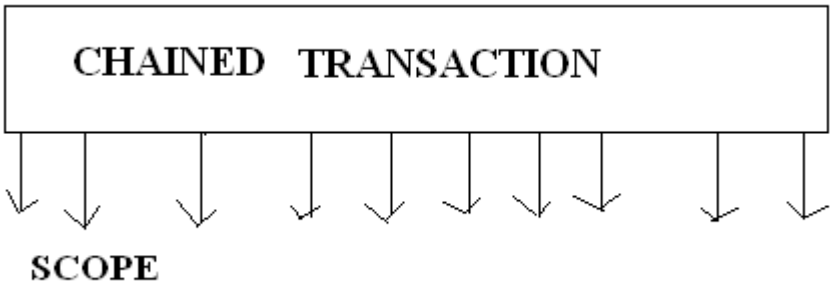
Flat transaction:-

- Transaction begins with a BEGIN_TRANSACTION
- Transaction on the database includes operations like read,inserts, updates and deletions are performed
- If transaction succeeds then a COMMIT is done else rolledback using ROLLBACK



Chained transaction

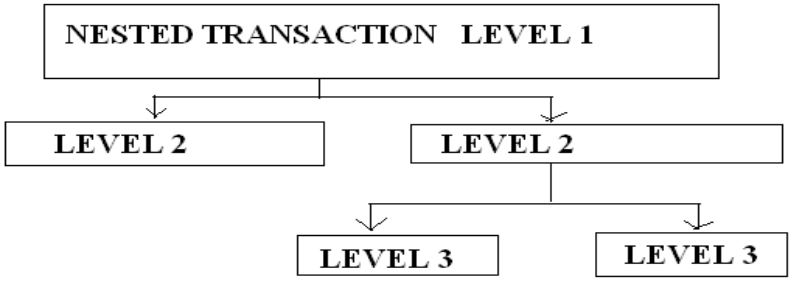
- Main transaction can contain several mini transactions so that each mini transaction must COMMIT after its successful completion
- Main transaction ends only after all its mini transactions get completed



Nested transaction

- Transaction allows a hierarchical transactional relationship between the master and the sub transaction
- In case of failure of a sub transaction the immediate higher level transaction can detect it and can make alternative approaches to re-do the sub transaction
- Distributed environment transaction are nested
- To preserve the ACID properties of a flat transaction in a distributed environment two phase commit protocol is used

- One of the participating server acts as a transaction coordinator. During the voting phase in response to the coordinators intimation to prepare to commit, then servers decide whether they can commit.
- The other servers have successfully completed their transaction each one them will send a request to perform a commit.
- In case of failure notice is sent to the coordinator. The coordinator evaluates whether all the responses are a success. If success the it will start the completion phase where it sends the COMMIT message to all the other servers.
- ROLL BACK message is sent to all the other servers so that the transaction is performed either completely or nothing at all.



2.7 Concurrency in server objects

In basic variant of the broker pattern

- Server enters a loop and waits for incoming request.
- Server responds to the requests sequentially.
- Broker sharing address space with the server, forwards each of incoming requests to a separate thread in the server (pool of available thread can be created for each request)
- Server is multithreaded then server object may be subject to invocation of several of its methods simultaneously
- Synchronization tools have to be applied in code of the server object in order to avoid race conditions
- Several threads running in the server may call the Broker at the same
- To register a newly created object. Broker supporting this concurrency is called multithreaded Broker

2.8 Event driven programming

- Event driven programming allow programs to respond to events and inputs
- Remote reference can be passed easily as parameter and it makes employing event based programming style without call back constructs
- Distributed object define some abstractions to support event driven programming style

- Components based on Observer pattern (Publisher – Subscriber)

Observer pattern (Publisher – Subscriber)

- Observer pattern is also known as the *Publish/Subscribe* pattern
- The process of registration is called *subscription*, and the process of notification is called *publication*
- One dedicated component takes the role of publisher and components dependent on changes in the publisher are its subscriber
- Publisher maintains a registry of currently –subscribe interface offered by the publisher
- Achieved by one-way propagation of changes
- Publisher announces interfaces (its listeners have to honor) and method of interface called to report on an event
- Listener object (subscriber) can subscribe to an event source (Publisher) to notified about and event occurrence (asynchronous signal)
- Listener subscribe dynamically
- In Model-View-Controller (MVC) pattern, the *Model* is an object that manages the data and behavior of some application domain: it is the subject in the Observer pattern. *Views* register with the Model as observers. Whenever the *Controller* makes a change to the Model, the Model notifies its registered observers (the Views) that it (the Model) has changed
- Event-driven architecture is an architectural style that builds on the fundamental aspects of event notifications to facilitate immediate information dissemination and reactive business process execution

Observer Pattern - Observer pattern is widely used in doing event-driven programming with GUI frameworks

- Java techniques for registering GUI event handlers
- event-handling aspects of the GUI, the java.awt.event package provides a number of different types of event-object:

ActionEvent	InvocationEvent
AdjustmentEvent	ItemEvent
ComponentEvent	KeyEvent
ContainerEvent	MouseEvent
FocusEvent	MouseWheelEvent
InputEvent	PaintEvent
InputMethodEvent	TextEvent

Many event-driven applications are *stateless*. This means that when the application finishes processing an event, the application hasn't been changed by the event. As event sources publish these notifications, event receivers can choose to listen to or filter out specific events, and make proactive decisions in near real-time about how to react to the notifications. For example, update customer records in internal systems as a result of a new customer registration on the Website, update the billing and order fulfillment systems as a result of an order checkout, update customer account as a result of a back office process completion, transforming and generating additional downstream events, and so on.

In an event-driven architecture, information can be propagated in near-real-time throughout a highly distributed environment, and enable the organization to proactively respond to business activities. Event-driven architecture promotes a low latency and highly reactive enterprise, improving on traditional data integration techniques such as batch-oriented data replication and posterior business intelligence reporting. Modeling business processes into discrete state transitions (compared to sequential process workflows) offers higher flexibility in responding to changing conditions, and an appropriate approach to manage the asynchronous parts of an enterprise.

Handlers Design Pattern

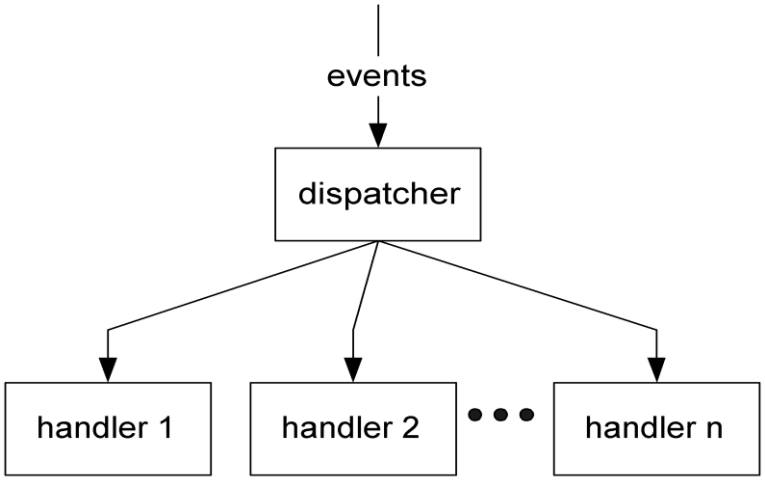


Fig. Handlers pattern

- a stream of data items called *events* (transactions)
- a *dispatcher* (transaction center)
- a set of *handlers*

The job of the dispatcher is to take each event that comes to it, analyze the event to determine its *event type*, and then send each event to a handler that can handle events of that type. The dispatcher must process a stream of input events, so its logic must include an *event loop* so that it can get an event, dispatch it, and then loop back to obtain and process the next event in the input stream.

Some applications (for example, applications that control hardware) may treat the event stream as effectively infinite. But for most event-handling applications the event stream is finite, with an end indicated by some special event — an end-of-file marker, or a press of the ESCAPE key, or a left-click on a CLOSE button in a GUI. In those applications, the dispatcher logic must include a *quit* capability to break out of the event loop when the *end-of-event-stream* event is detected.

In some situations, the dispatcher may determine that it has no appropriate handler for the event. In those situations, it can either discard the event or raise (throw) an exception. GUI applications are typically interested in certain kinds of events (e.g. mouse button clicks) but uninterested in others (e.g. mouse movement events). So in GUI applications, events without handlers are typically discarded. For most other kinds of applications, an unrecognized event constitutes an error in the input stream and the appropriate action is to raise an exception.

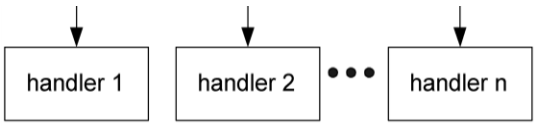
Here, pseudo-code for a typical dispatcher that shows all of these features:

- the event loop,
- the *quit* operation,
- the determination of event type and the selection of an appropriate handler on the basis of that type, and the treatment of events without handlers.

```
do forever:    # the event loop

   get an event from the input stream

   if   event type == EndOfEventStream :
        quit # break out of event loop

   if   event type == ... :
        call the appropriate handler subroutine,
        passing it event information as an argument

   elif event type == ... :
        call the appropriate handler subroutine,
        passing it event information as an argument

   else:   # handle an unrecognized type of event
        ignore the event, or raise an exception
```

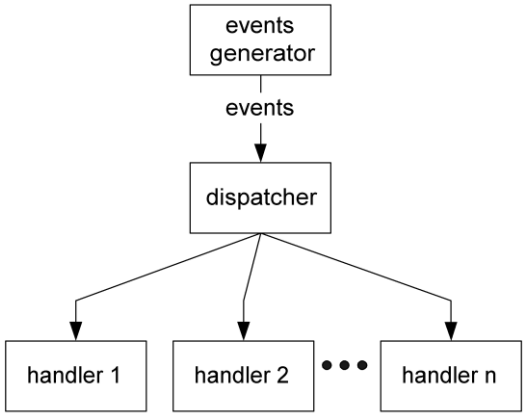
***Headless Handlers* Pattern**

There are a few variants on the Handlers pattern. One of them is the *Headless Handlers* pattern. In this pattern, the dispatcher is either missing or not readily visible. Taking away the dispatcher, all that remains is a collection of event handlers.



***Extended Handlers* Pattern**

Another variant is the *Extended Handlers* pattern. In this variant, the pattern includes an *events generator* component that generates the stream of events that the dispatcher processes.



Event Queue

In some cases, the dispatcher and its handlers may not be able to handle events as quickly as they arrive. In such cases, the solution is to buffer the input stream of events by introducing an *event queue* into the events stream, between the events generator and the dispatcher. Events are added to the end of the queue as fast as they arrive, and the dispatcher takes them off the front of the queue as fast as it is able.

GUI applications typically include an event queue. Significant events such as mouse clicks may require some time to be handled. While that is happening, other events such as mouse-movement events may accumulate in the buffer. When the dispatcher becomes free again, it can rapidly discard the ignorable mouse-movement events and quickly empty the event queue.

Potential Benefits

- Effective data integration

  In synchronous request-driven architectures, focus is typically placed on reusing remote functions and implementing process-oriented integration. Consequently, data integration is not well supported in a process-oriented model, and is often an overlooked aspect in many SOA implementations. Event-driven architecture is inherently based on data integration,

where the events disseminate incremental changes in business state throughout the enterprise; thus it presents an effective alternative of achieving data consistency, as opposed to trying to facilitate data integration in the process layer.

- Reduced information latency

  As distributed systems participate in an event-driven architecture, event occurrences are enabled to ripple throughout the connected enterprise in near real-time. This is analogous to an enterprise nervous system where the immediate propagation of signals and notifications form the basis of near real-time business insight and analytics.

- Enablement of accurate response

  As business-critical data propagates throughout the enterprise in a timely manner, operational systems can have the most accurate, current view of the state of business. This enables workgroups and distributed systems to provide prompt and accurate business-level responses to changing conditions. This also moves us closer to being able to perform predictive analysis in an ad hoc manner.

- Improved scalability

  Asynchronous systems tend to be more scalable than synchronous systems. Individual processes block less and have reduced dependencies on remote/distributed processes. Also, intermediaries can be more stateless, reducing overall complexity. Similar points can be made for reliability, manageability, and so forth.

- Improved flexibility

  Discrete event-driven processes tend to be more flexible in terms of the ability to react to conditions not explicitly defined in statically structured sequential business processes, as state changes can be driven in an ad hoc manner. And as mentioned earlier, the higher level of decoupling between connected systems means changes can be deployed more independently and thus more frequently.

- Improved business agility

  Event-driven architecture promises enhanced business agility, as it provides a closer alignment with operational models, especially in complex enterprise environments consisting of diverse functional domains and requiring high degrees of local autonomy. Also, business concepts and activities are modeled in simpler terms with responsibilities appropriately delegated to corresponding owners. This creates an environment where decisions and business processes in one area are less dependent on the overall enterprise environment,

compared to synchronous request-driven architecture's centralized view of logically coupled, sequentially structured business processes, where one component is often dependent on a number of distributed components for its processing. The reduced dependencies allow technical and functional changes to be planned and released more independently and frequently, achieving a higher level of IT agility and, hence, business agility.

Unit- III

Benefits of java programming with CORBA – CORBA overview – OMG – OMA – Object model – ORB structure – OMG, IDL, ORB and object interface – POA – Language mapping – Mapping – ORB run time system – Discovering services (Naming, Trading) – Advanced features (DSI – DII Interoperability, DII and DSI, IR. Overview of java ORBs – First Java ORB application – OMG IDL to Java, Interface repository) – CORBA events – Practical applications.

3.1 Benefits of java programming with CORBA

a) Object Oriented programming Language:
- Java ORBs provide the same functionality as any other ORB.
- Language bindings offered by current ORB products are C++,C, COBOL, ADA and Small talk.
- Java provides a cleaner approach to object-oriented programming than C++ with memory management responsibilities, no pointer, less confusing syntax and simpler method resolution rules.
- Supports features such as automatic garbage collection and integrated thread support.

b) Portability of application across platforms
- Significant over other programming languages byte-code set will be usable on any platform without porting

c) Web Integration:
- Java integrates well with both web browsers and web servers, and thus provides excellent support for the development of web-based applications. For CORBA application that need to be integrated with web infrastructure, java is the natural choice
- Java servlets and Java Server Pages(JSP) used to creating dynamic web content. The architecture and design of Java Servlets is superior to the web server's traditional Common Gateway Interface (CGI) or other proprietary APIs. Servlets(JSP are eventually compiled into servlets) can act as CORBA clients in a multitier architecture

d) Component Models:
- Java provides two different component modes, Java Beans and Enterprise Java Bean(EJB) with different application areas
- Using components is the increase in developer productivity (reuse & facilitating composition)

NOTE:

Java language binding for OMG IDL provides an application programmer with CORBAs high level distributed object paradigm

- Interfaces defined independently of implementations
- Access to objects implemented in other programming languages
- Access to objects regardless of their location (location transparency)
- Automatic code generation to deal with remote invocations
- Access to standard CORBA services and facilities

3.2 CORBA OVERVIEW

CORBA Stands for Common Object Request Broker Architecture. CORBA is a specification for creating distributed objects. CORBA is NOT a programming language and its architecture is based on the object model. CORBA promotes design of applications as a set of cooperating objects. OMG Object Model: object is defined as what the client could see.

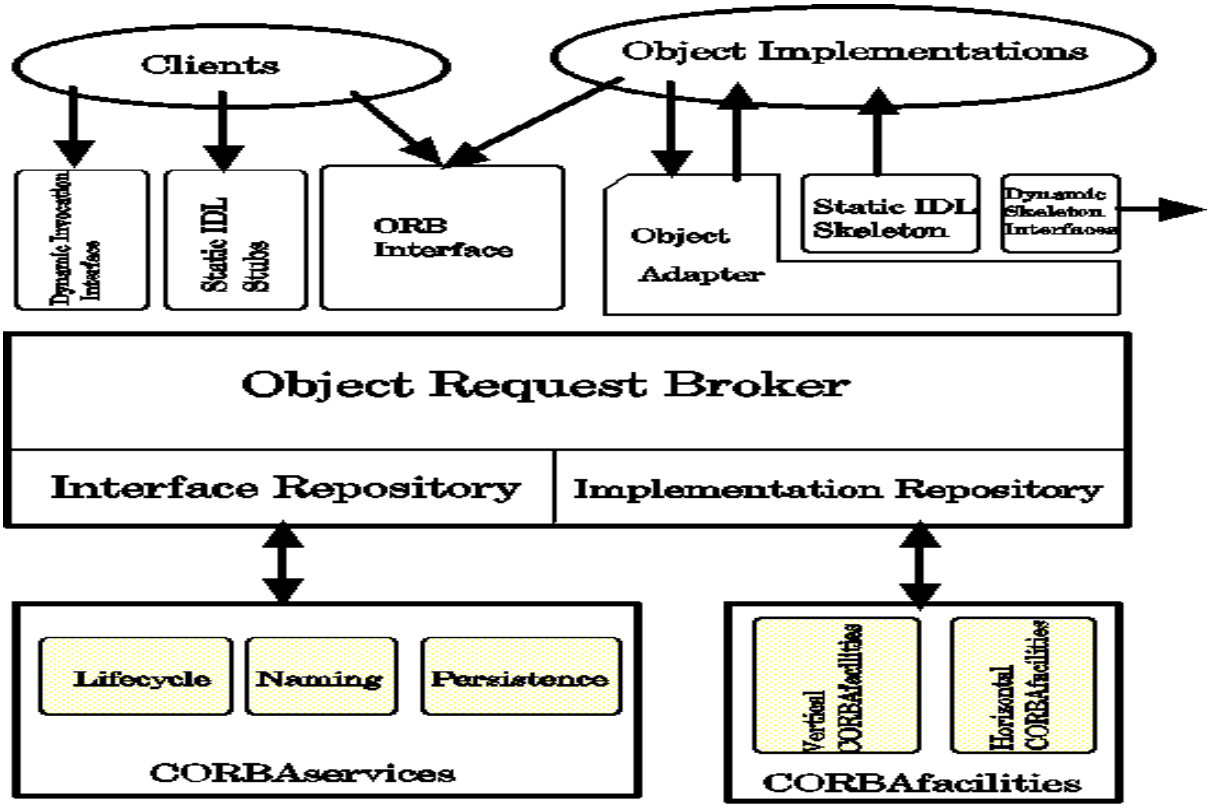


Figure 3.1: CORBA ARCHITECTURE

CORBA object request broker allow clients to invoke operations on distributed objects. The importance of CORBA standard is as follows:

a) Object Location: CORBA objects can be collocated with the client or distributed on a remote server, without affecting their implementation or use.

b) Programming Language: The language supported by corba include c, c++,java, cobal and smalltalk

c) OS platform: CORBA runs on many os platforms , including win32 ,unix and real time embedded system like lynxos.
d) Communication protocol and interconnects : The communication protocols and interconnects that CORBA can run on include TCP/IP , IPX/SPX, FDDI, ATM, Ethernet, fast Ethernet, embedded system backplanes, and shared memory.
e) Hardware : CORBA shields applications from side effects stemming from differences in hardware such as storage layots and data types size/ranges

CORBA Components

The CORBA specification is comprised of several parts :

a) An Object Request Broker (ORB)
b) Basic Object Adapter (BOA), Portable Object Adapter(POA)
c) An Interface Definition Language (IDL)
d) A Static Invocation Interface (SII)
e) A Dynamic Invocation Interface (DII)
f) A Dynamic Skeleton Interface(DSI)
g) Interface and implementation repositories
h) Programming language mappings
i) An Interoperability Spec(GIOP and IIOP)

Other documents form OMG describe common object services built upon CORBA services
e.g. , Event services, Name services, Lifecycle service

OBJECT:

- Object is a CORBA programming entity that consists of an *identity*, an *interface*, and an *implementation*, which is known as a *Servant*. an object is an instance of an interface definition language interface.
- The object is identified by an object reference, which uniquely names that instance across servers. An object id associates an object with its servent implementation , and is unique within the scope of an object adapter

Servant:

- Servant is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- Servants are implemented using one or more objects. Servants are typically implemented using functions and structures.

Client:

- Client is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Accessing a remote object should be a simple as calling an operation on a local object.

Object Request Broker (ORB)

- The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations.
- ORB makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

ORB Interface

- An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB.
- This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

CORBA IDL stubs and skeletons

- CORBA IDL stubs and skeletons serve as the ``glue'' between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler.

IDL Complier

- Compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

Dynamic Invocation Interface (DII)

- The DII allows clients to generate request at arun time. The flexiblity is useful when an application has no compile tine knowledge of the interface it is accessing.
- DII also allows clients to make differed synchronous calls, which are decouple the request and response portions of two way operations to avoid blocking the client until the serrvant responds.

Dynamic Skeleton Interface (DSI)

- This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing.
- The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

Interface repository:

- The Interface Repository provides run time information about IDL interfaces. It is possibble for a program to encounter an object whose interface was not known when the program was compiled and what operation are valid on the object and make invocations on it.
- The IR provides a common location to store additional information associated with interfaces ORB objects, such as stub/Skelton type libraries.

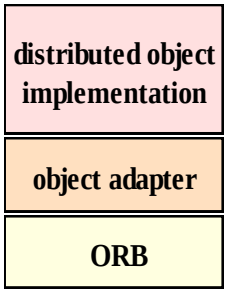
Implementation Repository:

- The Implementation Repository contains information that allows an ORB to activate servers to process servants. Most of the information in the implementation repository is specific to an ORB or OS environment.
- The implementation Repository provides common location to store information associated with servers, such as administrative control, resource allocation, security, and activation modes.

Object Adapter

- Object adapter describes the ORB with delivering requests to the object and with activating the object. An object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

In the basic architecture of CORBA, the implementation of a distributed object interfaces with the skeleton to interact with the stub on the object client side. As the architecture evolved, a software component in addition to the skeleton was needed on the server side: an object adapter.



- An object adapter simplifies the responsibilities of an ORB by assisting an ORB in delivering a client request to an object implementation (binding and polymorfism)
- When an ORB receives a client's request, it locates the object adapter associated with the object and forwards the request to the adapter.
- The adapter interacts with the object implementation's skeleton, which performs data marshalling and invoke the appropriate method in the object.

Object adapters are responsible for the following functions:

- Generation and interpretation of object references
- Demultiplexing request to servants
- Collaborating with Idl skeletons to invoke servant operations

- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations

3.3 OMG (Object Management Group)

Object Management Group (OMG) was formed in 1989. Its aims were:

- to make better use of distributed systems
- to use object-oriented programming
- to allow objects in different programming languages to communicate with one another
- Common Object Request Broker Architecture
- Created by Object Management Group (consortium of 700+ companies)
- Defines how distributed, heterogeneous objects can interoperate
- The object request broker (ORB) enables clients to invoke methods in a remote object

- CORBA is a specification of an architecture supporting this
  - CORBA 1 in 1990 and CORBA 2 in 1996.

3.4 **OMA (Object Management Architecture)**

OMA is a frame work within which all OMG adopted technology fits. Two fundamental models

(i) Core Object Model

- Abstract definition
- Details how ORB facilitates distributed application development

(ii) Reference Object Model

- ORB at center, interface definitions
- Frame work for future technology adoption

CORBA is an international standard for an Object Request Broker - middleware to manage communications between distributed objects. CORBA has been defined by the Object Management Group.

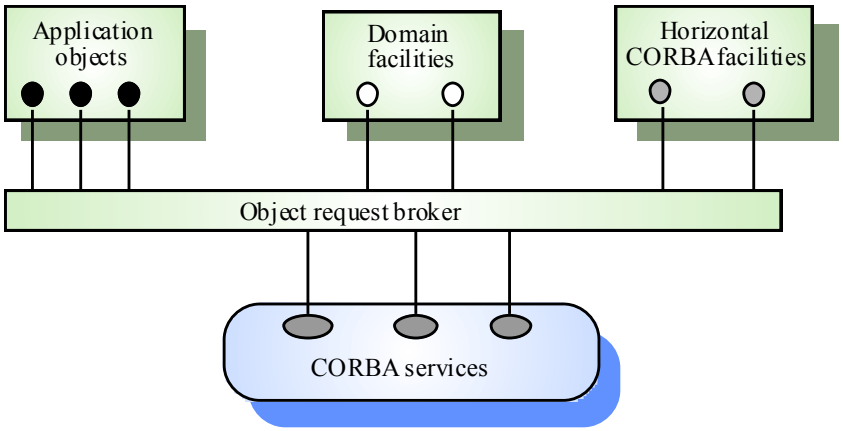


Figure 3.2 OMA architecture

Application structure

- Application objects
- Standard objects, defined by the OMG, for a specific domain e.g. insurance
- Fundamental CORBA services such as directories and security management
- Horizontal (i.e. cutting across applications) facilities such as user interface facilities

CORBA Services

- CORBA Services provides basic functionality, similar to the services that system library calls do in UNIX.
- Functions includes creating objects, controlling access to objects, keeping track of relocated objects and to consistently maintain relationship between objects.

Horizontal CORBA Facilities

- Horizontal CORBA Facilities sit between the CORBA services and Application objects.
- These components providing support across an enterprise and across business.
- Four facilities: the Printing Facilities, the Secure Time Facilities, the Internationalization Facilities, and Mobile Agent Facilities.

Domain(Vertical) CORBA Facilities

- Domain CORBA Facilities are the most exciting work at OMG.
- Define a standard interfaces for standard objects shared by companies within a specific vertical market(e.g. healthcare, manufacturing, finance).
- Now nine industries have their own OMG task force.

Application Objects

- Topmost part of the OMA hierarchy.
- Provide access to application objects that can invoke methods on remote objects through ORB. Application is built from a large number of basic object classes, new classes can be generated or specified provided by CORBA services.
- Standardization is not required.

Three Benefits of using OMA

1. Coding is quicker, so application can be deployed sooner
2. Applications designed around discrete services have better architecture
3. Many OMA implementations have enterprise characteristics built in: they're robust, and they scale.

Companies Using CORBA Today

- AT&T
- The Weather Channel
- Raytheon Company
- Chase Manhattan Bank
- Nokia Telecommunications

3.5 Object model

The OMA core object model provides some fundamental definitions of concepts that are extended by the CORBA specification.

Servants and Object References -The servant is a programming language entity containing code that implements the operations defined by an IDL interface definition, for example, a Java object. Object references are handles to objects. A given object reference will always denote a single object, but several distinct object references may denote a single object.

Types – object types are related in an inheritance hierarchy, with the type object at the root. Object type derived from another can be substituted for it. Object types may be specified as parameters and return types for operations, and they may be used as components in structured data types.

Interfaces – an interface is a description of the operations that are offered by an object.

Valuetypes and abstract interfaces – a valuetype like an interface, is a description of a set of operations.

Operation signature – Each operation has a signature, expressed in IDL, that contains an operation identifier(operation name), type of the value returned by the operation, direction (in,out, inout).

Attributes – an interface may contain attributes.

Exceptions – the standard IDL module, CORBA, contains declaration of 31 standard exceptions to address network, ORB, and operating system errors.

3.6 ORB structure

The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request.

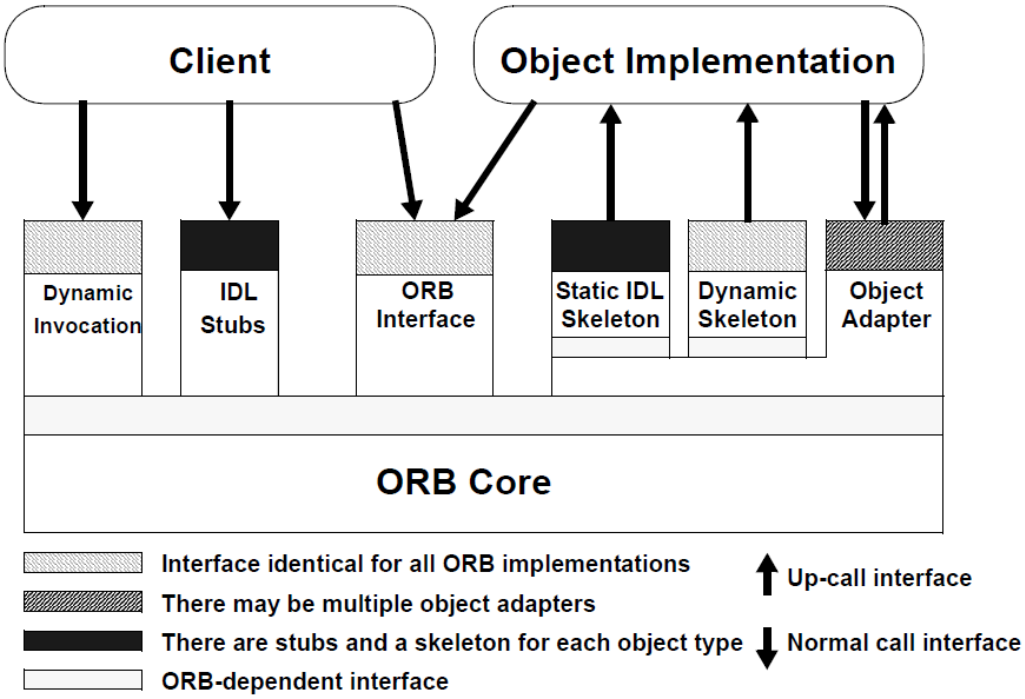


Figure 3.3 ORB structure

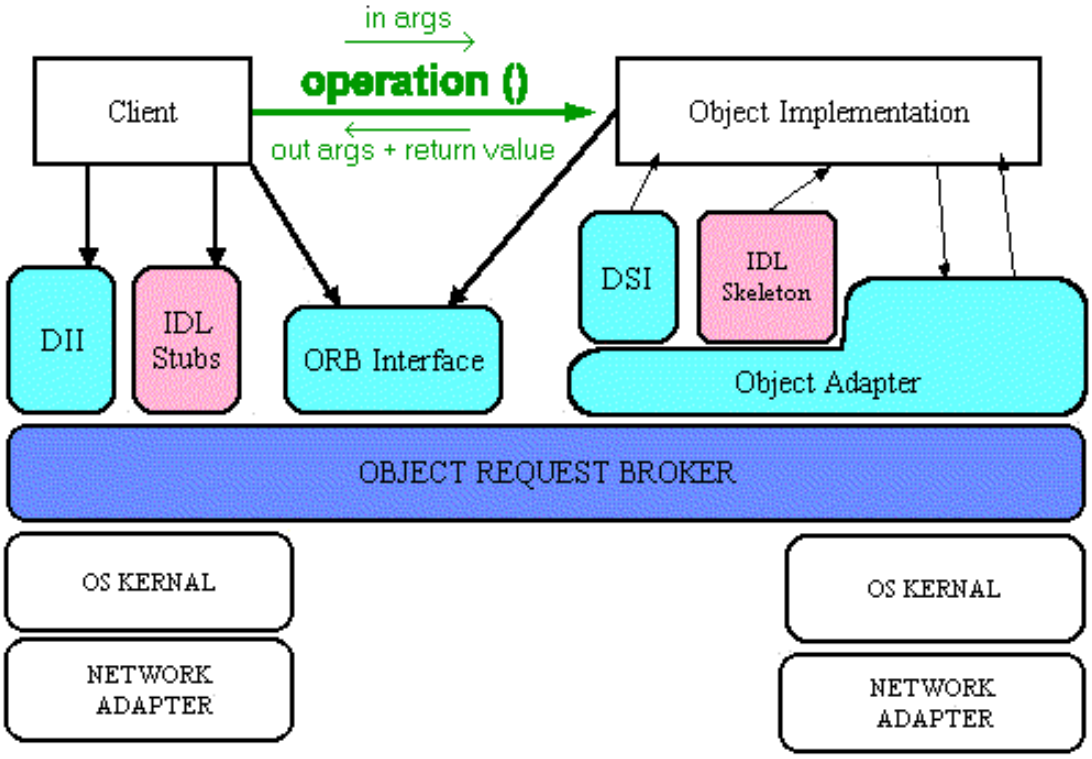


Figure 3.4 Client Request / Response through ORB

- To make a request, the Client can use the Dynamic Invocation interface (the same interface independent of the target object's interface) or an IDL stub (the specific stub depending on the interface of the target object). The Client can also directly interact with the ORB for some functions.
- The Object Implementation receives a request as an up-call either through the IDL generated skeleton or through a dynamic skeleton
- The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times.
- Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (IDL).
- Interfaces can be added to an Interface Repository service.
- This service represents the components of an interface as objects, permitting run-time access to these components.
- In any ORB implementation, the Interface Definition Language (which may be extended beyond its definition in this document) and the Interface Repository have equivalent expressive power.

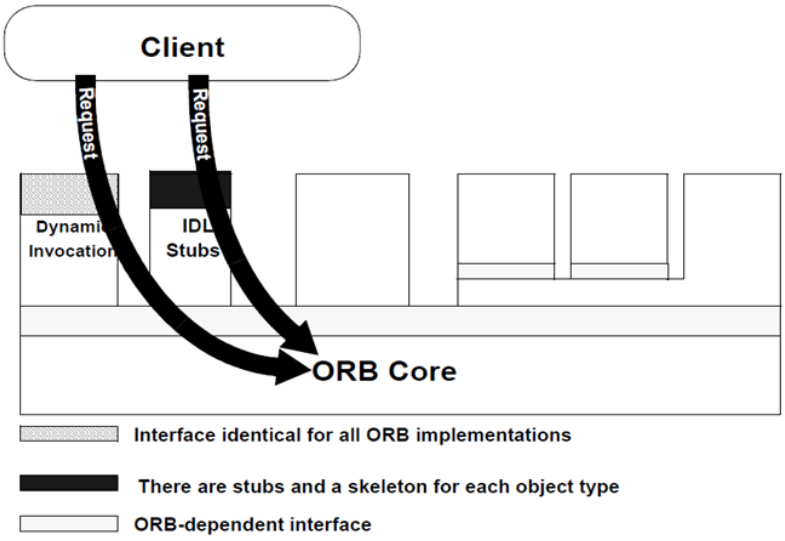


Figure 3.5 Client Request through ORB

- The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed.
- The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically

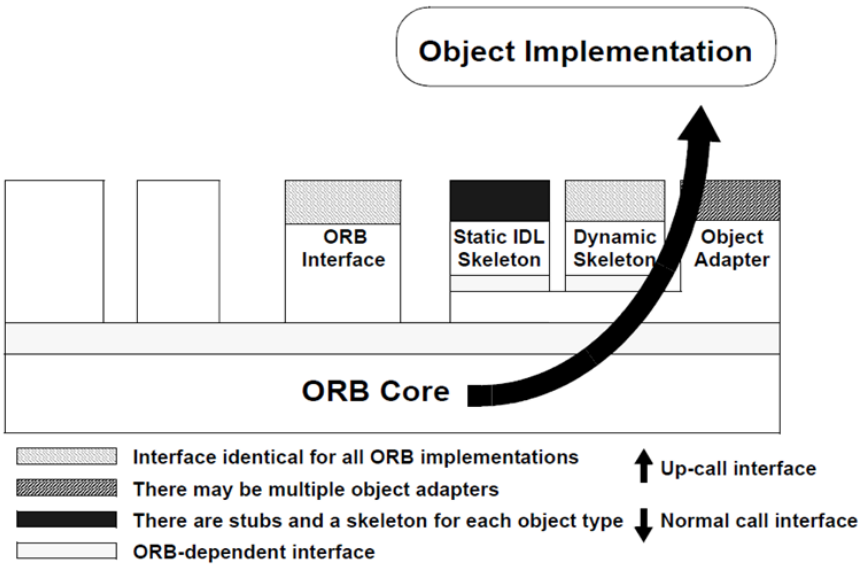


Figure 3.6 ORB to object communication

- The dynamic and stub interface for invoking a request.
- The ORB locates the appropriate implementation code, transmits parameters, and transfers control to the Object Implementation through an IDL skeleton or a dynamic skeleton.
- Skeletons are specific to the interface and the object adapter.
- In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter.
- When the request is complete, control and output values are returned to the client.

3.7 IDL and interface

- Stands for: Interface Definition Language
- A CORBA object is specified with interface
- interface: contract between client and server
- specified in a special declarative language
- Lexical rules are same as C++ with new keywords
- IDL mapping to programming languages (e.g. C++, Java, etc) are provided in specs

CORBA IDL Interfaces

- IDL interface provides a description of services available to clients
- IDL interface describes an object with:
  - Attributes
  - Methods with their signatures
  - Exceptions

- Inheritance information
- Type and constant definitions

CORBA IDL Interfaces

IDL Ground Rules

a) Case Sensitivity

- Identifiers are case sensitive
- names of identifiers in the same scope cannot differ in case only.
- Ex. in the myObject interface, IDL would not allow an operation named anOperation and another operation named anOPERATION to be simultaneously.

b) IDL Definition Syntax

- All definitions in IDL are terminated by a semicolon (;), much as they are in C, C++, and Java
- Definitions that enclose other definitions do so with braces. (modules & interfaces)
- When a closing brace also appears at the end of a definition, it is also followed by a semicolon it represents a module.

c) IDL Comments

- Both C-Style and C++-Style comments are allowed
- *// C++ comment allowed*

/ C – Style Comment allowed */*

d) **Use of the C Preprocessor**

- IDL assumes the existence of a C preprocessor to process constructs such as macro definitions and conditional compilation.
- *Ex: #ifdef….#endif, #include etc.*

e) Module

- **module** construct is used to group together IDL definitions that share a common purpose
- **module** declaration specifies the **module** name and encloses its members in braces
- Ex: *module Bank {*

```
interface Customer {
    …
  };
  interface Account {
  …
  };
  …
};
```

- The grouping together of similar interfaces, constant values, and the like is commonly referred to as ***partitioning*** and is a typical step in the system design process

- **Partitions** are also often referred to as modules or as packages

Primitive Types

IDL features a variety of primitive types. These types store simple values such as integral numbers, floating point numbers, character strings, and so on.

a) void

- This is analogous to the void type in C, C++ and Java
- This is useful for methods that don't return any value.

b) boolean

- Stores a boolean value either 0 (false) or 1(true)
- Depending on the programming language used, the IDL boolean type can map to an integral type (short int in C/C++) or to the language's native Boolean type (as in Java)
    - *boolean aBoolean;*

c) char and wchar

- This is analogous to char type in C/C++ and Java. And directly maps.
- Stores a single character value
- The char data type is an 8-bit quantity
    - *char aChar;*
- In version 2.1, wchar or wide character type is an 16-bit quantity

Floating Point Types

a) float

- The IDL float type represents an IEEE single-precision floating point value
- Analogous to C, C++, and Java
    - *float aFloat;*

b) double and ***long double***

- Represents an IEEE double-precision floating point value
- Corresponds to the double type in leading languages
    - *double aDouble*

Integer Types

- Unlike most familiar programming languages, IDL doesn't define a plain int type only short and long integer types.

a) long and ***long long***

- The IDL long type represents a 32-bit signed quantity, like C/C++'s int and Java's int
    - Ex: *long aLong;*

- In CORBA 2.1 the long type was added, which is a 64-bit signed quantity

b) unsigned long and ***unsigned long long***

- The unsigned long type in IDL is an unsigned version of the long type
  - Ex: *unsigned long anUnsignedLong;*
- CORBA 2.1 added the unsigned long long type, which is a 64-bit unsigned quantity.

c) short

- Represents a 16-bit signed quantity, as in C, C++, and Java
- It's range *-215 … 215-1.*
  - Ex: *short aShort;*

d) unsigned short

- Unsigned version of short
- Range *– 0…216-1*
  - Ex: *unsigned short aUnsignedShort;*

e) octet

- It is an 8-bit quantity that is not translated in any way during transmission.
- The similar type is available in Java as **byte**.
  - Ex: *octet aOctet;*

f) string

- Represents a collection of characters. Similar to C++'s **Cstring** and Java's **String** class.
- In 'C' character arrays are used as string.
- IDL supports fixed-length and variable-length strings.
  - Ex: *string aFixedLength[20]*

string aVariantLength;

The *const* modifier

- *const* values are useful for values that should not change.
- The scope of the *const* value is *interface or module.*
  - Ex: *const float pi = 3.14;*

IDL Keywords

any	**attribute**	**boolean**	**case**
char	**const**	**context**	**default**
double	**enum**	**exception**	**FALSE**
Fixed	**float**	**in**	**inout**
interface	**long**	**module**	**object**
octet	**oneway**	**out**	**raises**
readonly	**sequence**	**short**	**string**
struct	**switch**	**TRUE**	**typedef**
unsigned	**union**	**void**	**wchar**
wstring			

IDL Structure

```
module <identifier> {
  <type declarations>;
  <constant declarations>;
  <exception declarations;
   <interface definition>;
   <interface definition>;
};
```

IDL Structure (Modules)

- At the highest level, IDL definitions are packaged into module.
- A module is analogous to a package in Java
- Example:

```
module Bank {
  //body
};
```

IDL Structure (Interfaces)

- An IDL interface is the definition of an object
- IDL interfaces are analogous to Java interfaces
- An interface declares the operations that the CORBA object supports and its attributes

```
interface Account {
  //attributes
  // operations
};
```

IDL Structure (Interfaces)

- An interface may inherit from multiple interfaces
- It inherits all attributes and operations of its super-interfaces

```
interface JointSavingsAccount: JointAccount, SavingsAccount {
  // attributes
  // operations
};
```

IDL Structure (Attributes)

- They describe the variables (properties) of an interface
- An attribute can be **readonly** or read-write

```
interface Account {
  attribute string name;
```

```
    readonly attribute string sin;
    readonly attribute long accountNumber;
};
```

IDL Structure (Operations)

- They describe the methods of an interface
- They have a return type and parameters
- Parameters are flagged as:
  - **in**: parameter is passed from client to server
  - **out**: parameter is passed from server to client
  - **inout**: parameter is passed in both directions

```
void withdraw(in unsigned long amount);
void add(in long a, in long b, out long sum);
```

IDL Structure (Exceptions)

- IDL supports user-defined exceptions

```
exception InsufficientFunds {
  long currentBalance;
};
void withdraw(in unsigned long amount)
    raises (InsufficientFunds);
};
```

IDL to Java Mapping

IDL	Java
boolean	boolean
octet	byte
char	char
string	String
short	short
long	int
long lon	long
float	float

IDL	Java
double	double
fixed	BigDecimal

IDL - Constructed Types (User-defined data types)

Combine other types to enable the creation of user-defined data types.

a) enumerated type

- The enumerated type, ***enum*** allows the creation of types that can hold one of a set of predefined value specified by the ***enum***.
- Although the identifiers in the enumeration comprise an ordered list, IDL does not specify the ordinal numbering for the identifiers.

- This is similar to enum in C and C++

Ex:

```
enum DaysOfWeek
{
        Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
    };
```

b) structure type

- Contains any no. of member values of disparate types
- Structures are useful in IDL because, unlike CORBA objects, structures are passed by value rather than by reference.
- In other words, when a ***struct*** is passed to a remote object, a copy of that struct's value is created and marshaled to the remote object.

Ex:

```
struct DateStruct
{
        short year,
        short month,
        short day,
        short hour,
        short minute,
        short second,
        short microsecond
};
```

c) union type

- Represents values of different types.
- Resembles a cross between a C/C++ union and a case statement.

Ex:

```
union MultipleType switch(long)
{
        case 1:
                short myShortType;
        case 2:
                double myDoubleType;
        case 3:
        default:
                string myStringType;
```

};

- Here, the parameter is called as ***discriminator***.
- A discriminator used in an IDL union is a parameter that determines the value used by the union. The constant values in the case statements must match the discriminator's type.

3.8 ORB Interface

Provides for API that provide a number of utilities:

- Object-to-string : converts object reference to a string
- String-to-object : converts a string reference to object
- get-interface : to find the location of interface repository for a given object
- get-implementation: to find the reference to implementation repository of an object

3.9 Portable Object Adapter (POA)

Object Adapter Overview

CORBAObject Adapter introduces thePortable Object Adapter (POA) and contrasts the POA withits predecessor, the Basic Object Adapter (BOA).

Object Adapter Functionality

A CORBA Object Adapter is responsible for: (a) generating object references, (b) activation and deactivation of servants, (c) demultiplexing requests to servants, and (d) collaborating with IDL skeletons to invoke servant operations. These responsibilities are described in detail below:

Generating object references:

An Object Adapter is responsible for generating object references for the CORBA objects registered with it. Object references identify a CORBA object and contain addressing information that allow clients to invoke operations on that object in a distributed system. Object Adapters cooperate with the communication mechanisms in the ORB Core and underlying OS to ensure that the information necessary to reach an object is present in the object reference. (IOR), which supports the Internet Inter-ORB Protocol (IIOP). An IOR contains the IIOP version, host name, and port

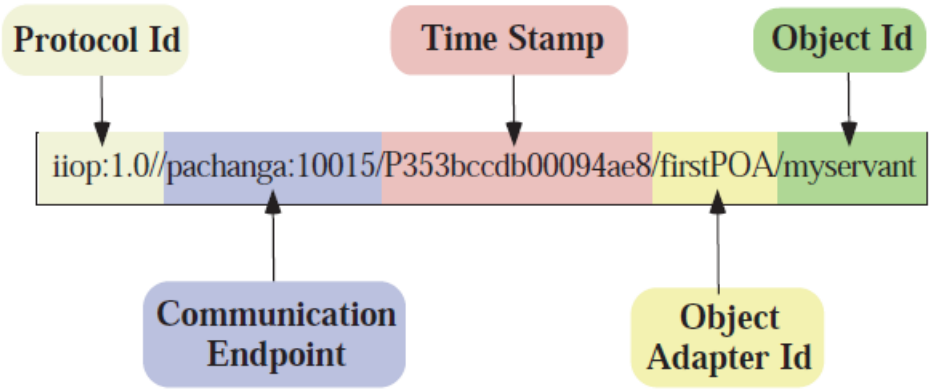


Figure 3.7 : Interoperable Object Reference

number that identifies a communication endpoint for the server

- process; some means to ensure uniqueness for certain types of
- IORs, *e.g.*, timestamps for *transient* IORs; the identity of the
- Object Adapter; and the identity of the CORBA object.

Activation and deactivation of servants:

Object Adapters can activate CORBA objects to handle client requests. To accomplish this, an Object Adapter can be programmed to create servants that handle requests for those objects. Similarly, Object Adapters can deactivate objects and can destroy their corresponding servants when they are no longer needed, *e.g.*, to reduce server memory consumption.

Demultiplexing requests to servants:

Object Adapters demultiplex CORBA requests to the appropriate servants. When an ORB Core receives a request, it collaborates with the Object Adapter through a private, *i.e.*, non-standardized, interface to ensure that the request reaches the proper servant. The Object Adapter parses the request to locate the Object Id of the servant, which it uses to locate the correct servant and invoke the appropriate operation on the servant.

Invoking servant operations:

The operation name is specified in the CORBA request. Once the Object Adapter locates the target servant, it dispatches the requested operation on the servant. Before the request is invoked on the servant, however, the Object Adapter uses an IDL skeleton to transform the parameters in the request into arguments. The skeleton then passes the demarshaled arguments as parameters to the intended servant operation.

Portable Object Adapter (POA)

The Portable Object Adapter (POA) is a standard component in the CORBA model recently specified by the OMG. The POA allows programmers to construct servants that are portable between different ORB implementations. Portability is achieved by standardizing the skeletons classes produced by the IDL compiler, as well as the interactions between the servants and the Object Adapter. The POA's predecessor

was the Basic Object Adapter (BOA). The BOA was widely recognized to be incomplete and underspecified. For instance, the API for registering servants with the BOA was unspecified. Therefore, different implementers made many interpretations and extensions to provide a complete ORB. These interpretations and extensions were incompatible with each other, however, and there was no simple upgrade to the BOA that made existing applications portable. The solution adopted by the OMG was to abandon the BOA and create a new Object Adapter that *was* portable. ORB implementors can maintain their proprietary BOA to support their current customer base. Existing programs continue to work and are supported by their ORB vendors. In the future, the OMG will no longer include the BOA with the CORBA specification.

POA Design Goals

The OMG's design goals for the Portable Object Adapter (POA) specification include the following:

a) **Portability:** The POA allows programmers to construct servants that are portable between different ORB implementations. Hence, the programmer can switch ORBs without having to modify existing servant code. The lack of this feature was a major shortcoming of the Basic Object Adapter (BOA).

b) **Persistent identities:** The POA supports objects with persistent identities. More precisely, the POA is designed to support servants that can provide consistent service for objects whose lifetimes span multiple server process lifetimes.

c) **Automation:** The POA supports transparent activation of objects and implicit activation of servants. This automation makes the POA easier and simpler to use.

d) **Conserving resources:** There are many situations where a server must support many CORBA objects. For example, a database server that models each database record as a CORBA object can potentially service hundreds of objects. The POA allows a single servant to support multiple Object Ids simultaneously. This allows one servant to service many CORBA objects, thereby conserving memory resources on the server.

e) **Flexibility:** The POA allows servants to assume complete responsibility for an object's behavior. For instance, a servant can control an object's behavior by defining the object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing code that will be executed in response to requests, and determining whether or not the object exists at any point in time.

f) **Behavior governed by policies:** The POA provides an extensible mechanism for associating policies with servants in a POA. Currently, the POA supports seven policies, such as threading, retention, and lifespan policies, that can be selected at POA creation time.

g) **Nested POAs:** The POA allows multiple distinct, nested instances of the POA to exist in a server. Each POA in the server provides a namespace for all the objects registered with that POA and all the child POAs that are created by this POA. The POA supports recursive deletes, *i.e.*, destroying a POA destroys its entire child POAs.

h) **SSI and DSI support:** The POA allows programmers to construct servants that inherit from (1) static skeleton classes (SSI) generated by OMG IDL compilers or (2) a Dynamic Skeleton Interface (DSI).

Clients need not be aware that a CORBA object is serviced by a DSI servant or an IDL servant. Two CORBA objects supporting the same interface can be serviced one by a DSI servant and the other with an IDL servant. Furthermore, a CORBA object may be serviced by a DSI servant during some period of time, while the rest of the time is serviced by an IDL servant.

POA Architecture

The ORB is an abstraction visible to both the client and server. In contrast, the POA is an ORB component visible only to the server, *i.e.*, clients are not directly aware of the POA's existence or structure. This section describes the architecture of the request dispatching model defined by the POA and the interactions between its standard components and the ORB Core. User-supplied servants are registered with the POA.Clients hold object references upon which they make requests, which the POA ultimately dispatches as operations on a servant. The ORB, POA, servant, and skeleton all collaborate to determine (1) which servant the operation should be invoked on and (2) to dispatch the invocation. a distinguished POA, called the Root POA, is created and managed by the ORB. The Root POA is always available to an application through the ORB initialization interface, resolve initial references. The application developer can register servants with the Root POA

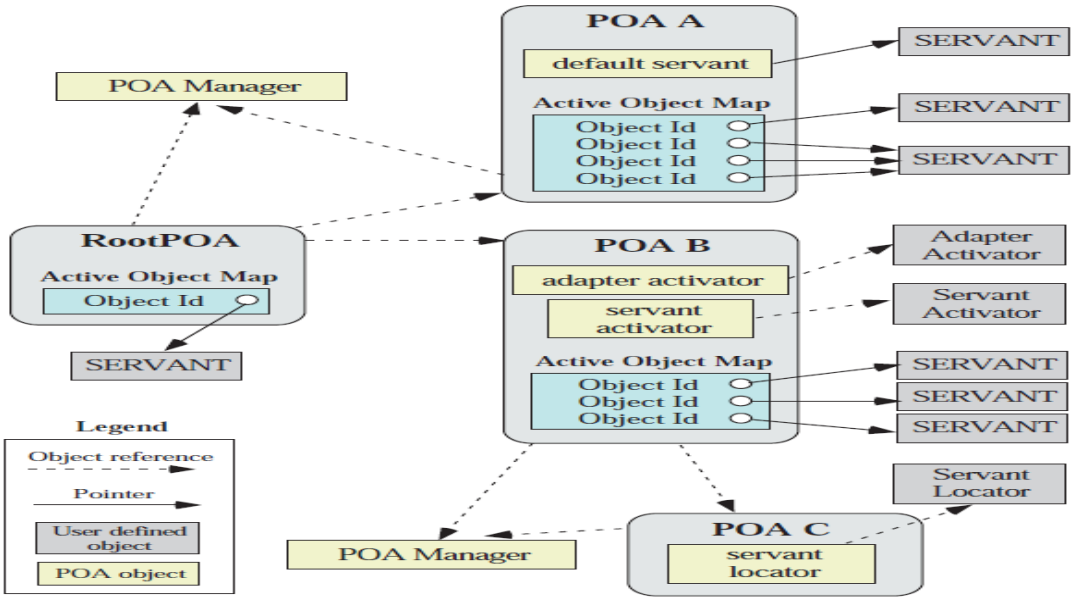


Figure 3.8 : POA ARCHITECTURE

A server application may want to create multiple POA to support different kinds of CORBA objects and/or different kinds of servant styles. For example, a server application might have two POAs: one supporting transient CORBA objectsand the other supporting persistent CORBA objects .A nested POA can be created by invoking the create POA factory operation on a parent POA. The server application contains three other nested POAs: A, B, and C. POA A and B are children of the Root POA; POA C is B's child. Each POA has an *Active Object Table* that maps Object Ids to servants.

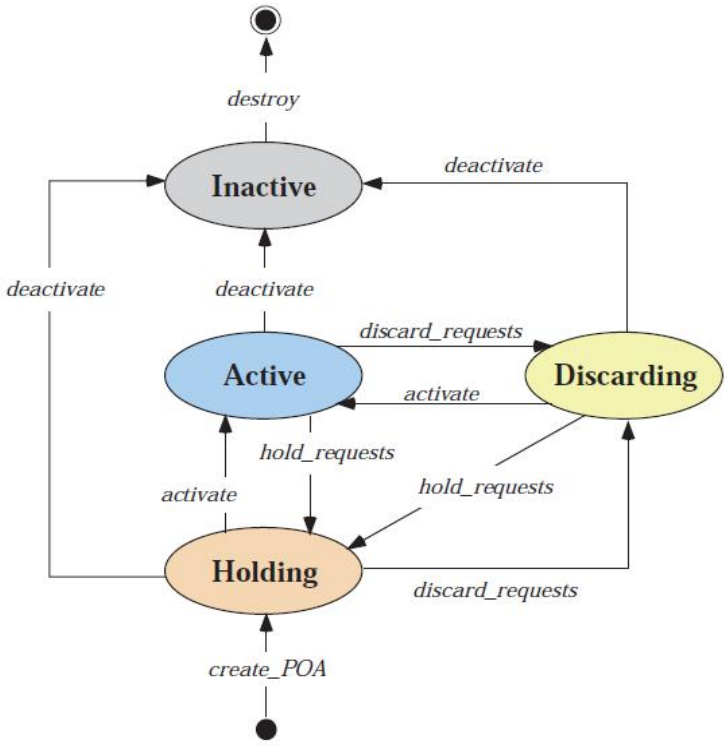


Figure 3.9: POA level activation and deactivation

Other key components in a POA are described below:

a) POA Manager:

A POA manager encapsulates the processing state of one or more POAs. By invoking operations on a POA manager, server applications can cause requests for the associated POAs to be queued or discarded. In addition, applications can use the POA manager to deactivate POAs. Figure shows the processing states of a POA Manager and the operations required to transition from one state to another.

b) Adapter Activator:

An adapter activator can be associated with a POA by an application. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not yet exist. The adapter activator can then decide whether or not to create the required POA on demand. For example, if the target object reference was created by a POA whose full name is /A/B/C and only POA /A and POA /A/B currently exist, the unknown adapter operation will be invoked on the adapter activator associated with POA /A/B. In this case, POA /A/B will be passed as the parent parameter and C as the name of the missing POA to the unknown adapter operation.

c) Servant Manager:

A servant manager is a locality constrainted servant that server applications can associate with a POA .The ORB uses a servant manager to activate servants on demand, as well as to deactivate servants. Servant managers are responsible for (1) managing the association of an object (as

characterized by its Object Id value) with a particular servant and (2) for determining whether an object exists or not. There are two types of servant managers: ServantActivator and ServantLocator

d) POA Policies:

The characteristics of each POA other than the Root POA can be customized at POA creation time using different *policies*. The policies of the Root POA are specified in the POA specification. The POA specification defines the following policies:

e) Threading policy:

This policy is used to specify the threading model used with the POA. A POA can either be single-threaded or have the ORB control its threads. If it is single-threaded, all requests are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code, *i.e.*, servants and servant managers, are invoked in a manner that is safe for code that is unaware of multi-threading. In contrast, if the ORB-controlled threading policy is specified, the ORB determines the thread (or threads) that the POA dispatches its requests in. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.

f) Lifespan policy:

This policy is used to specify whether the CORBA objects created within a POA are persistent or transient. Persistent objects can outlive the process in which they are created initially. In contrast, transient objects cannot outlive the process in which they are created initially. Once the POA is deactivated, use of any object references generated for a transient object will result in an CORBA::OBJECT NOT EXIST exception.

g) Object Id uniqueness policy:

This policy is used to specify whether the servants activated in the POA must have unique Object Ids. With the unique Id policy, servants activated with that POA support exactly one Object Id. However, with the multiple Id policy, a servant activated with that POA may support one or more Object Ids.

h) ObjectId assignment policy:

This policy is used to specify whether Object Ids in the POA are generated by the application or by the ORB. If the POA also has the persistent lifespan policy, ORB assigned Object Ids must be unique across all instantiations of the same POA.

i) Implicit activation policy:

This policy is used to specify whether implicit activation of servants is supported in the POA. A C++ server can create a servant, and then by setting its POA and invoking its this method, it can register the Servant implicitly and create an object reference in a single operation.

j) Servant retention policy:

This policy is used to specify whether the POA retains active servants in an *ActiveObject Map*. A POA either retains the associations between servants and CORBA objects or it establishes a new CORBA object/ servant association for each incoming request.

k) Request processing policy:

This policy is used to specify how requests should be processed by the POA. When a request arrives for a given CORBA object, the POA can do one of the following: _ *Consult its Active Object Map only* – If the Object Id is not found in the Active Object Map, the POA returns an CORBA::OBJECT NOT EXIST exception to the client. _ *Use a default servant* – If the Object Id is not found in the Active Object Map, the request is dispatched to the default servant (if available). _ *Invoke a servant manager* – If the Object Id is not found in the Active Object Map, the servant manager (if available) is given the opportunity to locate a servant or raise an exception. The servant manager is an applicationsupplied object that can incarnate or activate a servant and return it to the POA for continued request processing. Two forms of servant manager are supported: ServantActivator, which is used for a POA with the RETAIN policy, and ServantLocator, which is used with the NON RETAIN policy. Combining these policies with the retention policies described above provides the POA with a great deal of flexibility.

3.10 Discovering services (Naming, Trading)

Naming Services

One way for a server application to advertise an object is for the server to stringify an object reference—by calling object_to_string() write it to, say, a file. A client application could then read in the string, and call string_to_object() to turn it back into a proxy. This mechanism works fine *as long as* the client and server applications have access to a shared file system. This is likely to be the case if the client and server are running on the same computer or on the same local area network. However, it is unlikely that a wide area network will have a shared file system. For this reason, CORBA has matured over the years to support more geographically-scalable ways for a server to advertise an object. One such way, the Naming Service, is discussed in this chapter,

Basic Concepts

The white pages telephone book provides a mapping from a person's name to their contact details (address and telephone number). Likewise, the CORBA Naming Service provides a mapping from a (human-readable) name to an object's "contact details" (an IOR). However, that is where the analogy ends. The names in a telephone book are arranged alphabetically, while the names in the Naming Service are arranged as a hierarchy (similar to how the file system in UNIX or Windows is arranged as a hierarchy). Each level in the Naming Service hierarchy is called a *naming context* (while in a UNIX file system it is called a *directory* and in Windows it is called a *folder*). A naming context is itself a CORBA object (it is defined by the CosNaming::NamingContext interface). Because CORBA objects can be accessed regardless of their location, this means that a Naming Service hierarchy *could* be contained inside a single server process *or* the hierarchy could be spread over multiple Naming Service server processes. The concept of linking several

Naming Service server processes in this way is called *federation*. Many organizations do not make use of federation and instead use a monolithic Naming Service. However, some organizations *do* make use of a federated Naming Service. This is typically done to delegate administration responsibilities. For example, an organization might have a separate Naming Service process for each department or branch and then use federation to link these separate Naming Services into one logical unit that is the “company-wide” Naming Service.

```
#pragma prefix "omg.org"
module CosNaming {
   typedef string Istring;
   struct NameComponent {
      Istring  id;
      Istring  kind;
   };
   typedef sequence<NameComponent> Name;
   enum BindingType {nobject, ncontext};
   struct Binding {
      Name        binding_name;
      BindingType  binding_type;
   };
   typedef sequence <Binding> BindingList;

   interface BindingIterator {
      boolean next_one(out Binding b);
      boolean next_n(in unsigned long how_many,
                     out BindingList bl);
      void destroy();
   };
   interface NamingContext {
      void bind(in Name n, in Object obj) raises(...);
      void rebind(in Name n, in Object obj) raises(...);
      void bind_context(in Name n, in NamingContext nc)
                  raises(...);
      void rebind_context(in Name n,
                   in NamingContext nc) raises(...);
      Object resolve(in Name n) raises(...);
      void unbind(in Name n) raises(...);
      NamingContext new_context();
```

```
        NamingContext bind_new_context(in Name n)
                raises(...);
        void destroy() raises(...);
        void list(in  unsigned long   how_many,
                out BindingList     bl,
                out BindingIterator bi);
    };
    ... // interface NamingContextExt omitted for brevity
};
```

The functionality of the Naming Service is defined through the operations of its IDL interfaces. There are some operations that can be used to create/modify/delete a naming context hierarchy. There are other operations to *bind* (that is, advertise) an IOR in the Naming Service with a specified name, and yet other operations to *resolve* (that is, lookup) an IOR associated with a specified name.

It is common for an implementation of the Naming Service to provide some command-line utilities and/or a GUI tool that can be used to do administration tasks on the Naming Service, such as create/modify/delete a naming context hierarchy. Such administration functionality is implemented by invoking corresponding IDL operations on the Naming Service.

The CORBA specification for the Naming Service does not specify what *quality of service* should be offered by an implementation. Some implementations of the Naming Service hold the details of the *name→IOR* mappings in memory, which has the drawback that such details are lost whenever the Naming Service is killed and restarted, but has the benefit of not requiring access to persistent storage (which might be important in an embedded device). Other implementations of the Naming Service persist the *name→IOR* mappings in a file or database.Most, possibly all, implementations of CORBA have a bundled implementation of the Naming Service, which means that you do not have to purchase it separately. It is possible that the Naming Service bundled with the CORBA product you are using might have a quality of service that is unsuitable for your needs. If this is the case then you could discard that Naming Service and replace it with a Naming Service from a different vendor or even implement the Naming Service yourself. However, in practice, you are likely to be content with the quality of service offered by the Naming Service bundled with the CORBA product you decide to use. The ability to “rip out and replace” an implementation of the Naming Service (or another CORBA Service) is mentioned to illustrate the flexibility of an open standard (such as CORBA) that cleanly separates specification from implementation.

Imperfections in the IDL

Although the concepts in the Naming Service are simple, unfortunately the OMG made a few poor choices when defining the IDL types. This has resulted in confusion for some developers. Such confusion rapidly disappears when the motivation for the design choices are explained. This section explains the poor design choices with the aim of helping developers avoid unnecessary confusion. Readers who will not be doing development might prefer to skip this section.

The IDL types of the Naming Service are defined in the CosNaming module. Within this module, the NamingContext interface defines the operations that can be performed on a naming context. An application can connect to the root naming context of the Naming Service by passing "NameService" as a parameter to the resolve_initial_references() operation .The first piece of confusion likely to strike a programmer is that the parameter to this operation is "NameService" rather than "NamingService".

Most of the remaining confusion with the API of the Naming Service concerns the format of hierarchical names within the Naming Service. In hindsight, the OMG should probably have chosen to represent a hierarchical name as a string that uses "/" as a separator between naming contexts, for example, "path/in/naming/service". However, there were several perceived problems with this:

- The OMG wanted the hierarchical names to be expressed in multi-byte character strings rather than single-byte character strings. However, at the time that the Naming Service was being defined, the wstring type had not been introduced to IDL.
- Using "/" as a hierarchical separator would be familiar to people from a UNIX background. However, hierarchical file systems in other operating systems (for example, MS-DOS, VMS and MacOS) used different syntaxes for their hierarchical separator. There was no compelling reason to arbitrarily choose one separator syntax over another.
- Many file systems have the concept of a filename being composed of a *root-name* and an *extension*. For example, in the filename "foo.txt", the root-name is "foo" and the extension is "txt", with "." separating the two. Some people felt that similar flexibility might be useful in a hierarchical name in a Naming Service.

The OMG tried to resolve the string/wstring dilemma by introducing the following definition:

```
typedef string Istring;
struct NameComponent {
        Istring  id;   // denotes the "foo" part of "foo.txt"
        Istring  kind; // denotes the "txt" part of "foo.txt"
};
typedef sequence<NameComponent> Name;
```

The result is certainly flexible, but it is also over-engineered. Most people do not need all this flexibility and become frustrated with the complexity that it introduces to their applications. For example, consider an application that reads a string of the form "path/in/naming/service" from a runtime configuration file. The developers of such an application have to write their own function to convert the string into the CosNaming::Name format. One problem is that it is a waste of time for numerous developers in different organizations around the world to re-invent the wheel in writing such a utility function. Another problem is that each of these developers must choose what hierarchical separator they want and the separator between the id and kind fields of NameComponent. As it turns out, most developers choose to use UNIX-style separators of "/" and ".".

```
#pragma prefix "omg.org"
module CosNaming {
   ...
   interface NamingContextExt : NamingContext {
      typedef string StringName;
      StringName to_string(in Name n) raises(...);
      Name to_name(in StringName sn) raises(...);
      Object resolve_str(in StringName sn) raises(...);
      ...
   };
};
```

Hindsight is a wonderful thing, and a few years later the OMG decided to simplify the complexity of hierarchical names by defining a new version of the Naming Service. IDL does not have a versioning mechanism, so the simplified Naming Service was defined by defining a sub-type that inherits from NamingContext. The to_string() and to_name() operations convert between the CosNaming::Name and "path/in/naming/service" formats. As an extra convenience, clients can call resolve_str() to resolve (that is, lookup) an IOR from the Naming Service without having to convert "path/in/naming/service" into CosNaming::Name format. However, there is no similar utility operation defined for the bind() or rebind() operations that servers use to advertise an IOR in the Naming Service.The final area of potential confusion for developers is the list() operation, An out parameter of this operation provides a *non-recursive* listing of the entries in a naming context. Because the listing is non-recursive, the binding_name field of each Binding *should be* of type NameComponent. The accidental use of type Name makes some developers mistakenly think that this operation provides a recursive listing.

Once developers are aware of these imperfections, use of the Naming Service is straightforward. Certainly the imperfections of this API seem minor in comparison to imperfections in the APIs of some other (non-middleware) systems.

```
try {
   instructions1 = "name_service#path/in/Naming/Service";
   instructions2 = "file#/path/to/file.ior";
   obj1 = importObjRef(orb, instructions1);
   exportObjRef(orb, obj2, instructions2);
} catch(ImportExportException ex) {
   cout << ex << endl;
}
```

Trading Service:

CORBA provides several ways for a server to advertise an object reference. The types are Naming Service and trading service. In the Naming Service is compared to the white pages telephone book, the trading service is compared to the yellow pages telephone book. The yellow pages contains numerous *advertisements o*rganized into different *categories* (such as *Builders*, *Plumbers* and *Restaurants*). The Trading Service contains numerous *service offers* that are organized by their *service offer types*. Each advertisement in the yellow pages provides contact details (address and telephone number) for a company along with a description of the company. Similarly, each *service offer* in the Trading Service provides contact details along with a description of the service offered by the object.

- The Service Type Repository Interface
- The Register Interface
- The Lookup Interface
- Other Capabilities of the Trading Service
- Using the Trading Service
- Quality of Service

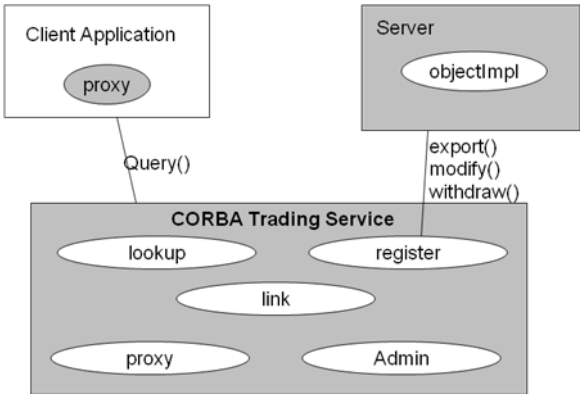


Figure 3.10 CORBA Trading services

The Service Type Repository Interface

There are several simplifications in this interface and the interfaces shown later in this chapter:

- The raises clause on operations has been omitted.
- Some typedef statements have been removed. For example, the name parameter to add_type() is shown as being of type string. In the real IDL definition, it is actually a typedef of a typedef of a string.
- Some of the parameters of operations and fields of structs are shown as being of an *anonymous sequence* type, such as sequence<string>. Use of anonymous sequences is illegal for parameter types and is deprecated for the types of fields in structs. The use of these anonymous types was done in order to avoid using extra typedef statements, and so to make the IDL listings more concise.

```
module CosTradingRepos {
  interface ServiceTypeRepository {
    enum PropertyMode {
    PROP_NORMAL,
    PROP_READONLY,
    PROP_MANDATORY,  PROP_MANDATORY_READONLY};
    struct PropStruct {
       string          name;
       CORBA::TypeCode  value_type;
       PropertyMode     mode;
    };
    typedefsequence<PropStruct> PropStructSeq;
    struct IncarnationNumber {
       unsigned long  high;
       unsigned long  low;
    };
    struct TypeStruct {
       string          name;
       PropStructSeq     props;
       sequence<string>  super_types;
       boolean          masked;
```

```
    IncarnationNumber incarnation;
  };
  ...
  readonly attribute IncarnationNumber incarnation;
  IncarnationNumber add_type(
          in string      name,
          in string      if_name,
          in PropStructSeq  props,
          sequence<string>  super_types)
              raises(...);
  void remove_type(in string name) raises(...);
  void mask_type(in string name) raises(...);
  void unmask_type(in string name) raises(...);
  ...
 };
};
```

The Trading Service equivalent of a “category” is a CosTradingRepos::ServiceTypeRepository::TypeStruct, although this is normally referred to as a *service offer type*. The ServiceTypeRepository interface is used to define service offer types.

The add_type() and remove_type() operations

The add_type() operation is used to define a new service offer type. The name parameter specifies the name of the category. The props parameter specifies a sequence of properties that are associated with the service offer type. The ServiceTypeRepository does not place any restriction on the types of properties. The super_types parameter lists the names of parent service offer types. A service offer type inherits the properties of its parents and is allowed to impose more restrictions on inherited properties. The remove_type() operation removes a service offer type.

The mask_type() and unmask_type() operations

The mask_type() operation is used to *mask* (hide) a service offer type. This has two uses. One use is to deprecate a service offer type, so that the Trading Service will not accept any more *service offers* (advertisements) for the specified service offer type. Another use is to indicate that the service offer type is an *abstract base type* that cannot be instantiated directly, but from which other service offer types can inherit. The unmask_type() operation unmasks a type that was previously masked with mask_type().

The Register Interface

When a service offer type has been defined, it is then possible for applications to actually *export* (advertise) an object reference in the Trading Service.

```
module CosTrading {
  struct Property {
    string  name;
    any     value;
  };
  typedef sequence<Property> PropertySeq;
  struct Offer {
    Object       reference;
    PropertySeq  properties;
  };
  typedef sequence<Offer> OfferSeq;
  interface Register
        : TraderComponents, SupportAttributes
  {
    string export(
              in Object       reference,
              in string       type,
              in PropertySeq  properties) raises(...);
    void withdraw(in string offer_id) raises(...);
    void modify(
              in string            offer_id,
              in sequence<string>  del_list,
              in PropertySeq       modify_list
           ) raises(...);
    ...
  };
};
```

The export() and withdraw() operations

The export() operation is used to create a *service offer*, that is, an advertisement for an object. Parameters to this operation specify an object reference, the service offer type that it matches, and its properties. The supplied properties must match those specified by the service offer type. The return value from export() is a

unique string that denotes an offer id. This offer id can later be used to *withdraw* the advertisement or *modify* some of its properties. The withdraw() operation is used to withdraw (that is, delete) a service offer.

The modify() operation

The modify() operation is used to delete or modify some properties associated with a service offer. An exception is thrown if an attempt is made to delete a mandatory property. Likewise, an exception is thrown if an attempt is made to modify a read only property.

The Lookup Interface

The Lookup interface has one operation, called query(). This operation is used to retrieve service offers (advertisements) from the Trading Service that match a specified constraint.

```
module CosTrading {
  struct Property {
    string  name;
    any     value;
  };
  typedef sequence<Property> PropertySeq;
  struct Offer {
    Object reference;
    PropertySeq properties;
  };
  typedef sequence<Offer> OfferSeq;
  interface OfferIterator {
    boolean next_n(in  unsigned long  n,
                   out OfferSeq       ids) raises(...);
    ...
  };
  interface Lookup
        : TraderComponents, SupportAttributes,
          ImportAttributes
  {
    enum HowManyProps {none, some, all};
    union SpecifiedProps switch (HowManyProps) {
      case some:  sequence<string>  prop_names;
    };
    ...
    void query(
              in string         service_type_name,
              in string         constraint,
```

```
            in string          preference,
            ...
            in SpecifiedProps  desired_props,
            in unsigned long   how_many,
            out OfferSeq       offers,
            out OfferIterator  offer_iter,
            ...) raises(...);
  };
};
```

3.11 Advanced features (DSI – DII Interoperability, DII and DSI, IR.

CORBA IDL stubs and skeletons - CORBA IDL stubs and skeletons serve as the **glue** between the client and server applications, respectively, and the ORB.

- The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler.
- The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

Dynamic Invocation Interface (DII) - This interface allows a client to directly access the underlying request mechanisms provided by an ORB.

- Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in.
- This allows clients to use objects discovered at runtime
- Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.

The dynamic invocation of a method

- To the object implementation the request looks the same whether it was dynamically or statically invoked.
- To dynamically invoke an object method the client follows four simple steps:

1) identifies the object to invoke,
2) retrieves the objects interface (via an ORB get_interface request),
3) constructs the request (via an ORB create_request call),
4) invokes the request and receives the results.

Dynamic Skeleton Interface (DSI) - This is the server side's analogue to the client side's DII.

- The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing.

  - The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.
- This is the server-side equivalent of the DII.
- The DSI allows server requests to be invoked on object implementations that do not have a static skeleton.
- Dynamic skeletons are very useful for providing bridges between ORBs for inter-orb requests.
  - to implement dynamic services or dynamically changing services
  - run-time binding of new services using scripting languages and interpreters
  - to redirect messages to objects for which a compiled idl-interface is not known

Interface Repository (IR)

  - Runtime database of interfaces of all registered objects
  - Supports the query of, registration of, and update of interfaces of objects mediated by the ORB
  - Repository IDs - uniquely and globally identify interface repository across multivendor ORBs

Implementation Repository

  - Keeps track of classes and run-time instances of server objects supported by the ORB.
  - Allows for mechanisms to support audit trails, trace information and security related infrastructure.

Building CORBA Applications

- Steps to build a CORBA application:
  1. Define the remote interface
  2. Compile the remote interface
  3. Implement the server
  4. Implement the client
  5. Start the applications
- Define the remote interface
  1. Define the interface for the remote object using the OMG's Interface Definition Langauge (IDL).
- Compile the remote interface
  1. Compile the IDL and map the interface to the designated language.
- Implement the server
  1. Use the skeletons it generates to develop the server application.
  2. Implement the methods of the remote interface.
  3. The server code includes a mechanism to start the ORB and wait for invocation from a remote client.

- Implement the client
    1. Use the stubs generated to develop the client application.
    2. The client code builds on the stubs to start its ORB, look up the server using the name service, obtain a reference for the remote object, and call its method.
- Start the applications
    1. Start the name service, then start the server, then run the client.

First Java ORB application

The steps involved:

- Define an interface
- Map IDL to Java (idlj compiler)
- Implement the interface
- Write a Server
- Write a Client
- Run the application

Example: a step-by-step Hello example

Step 1: define the interface

Hello.idl

```
module HelloApp {
  interface Hello {
    string sayHello();
  };
};
```

Step 2: map Hello.idl to Java

Use the idlj compiler (J2SE 1.3)

```
idlj –fall Hello.idl
```

This will generate:

_HelloImplBase.java (server skeleton)
HelloStub.java (client stub, or proxy)
Hello.java
HelloHelper.java, HelloHolder.java
HelloOperations.java

Step 3: implement the interface

Implement the servant:

```
import HelloApp.*;
class HelloServant extends _HelloImplBase {
   public String sayHello() {
```

```
        return “\nHello There\n”;
      }
    }
```

Step 4: implement the server

Import statements:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
class HelloServer {
  public static void main(String argv[]) {
    try {
```

Step 4: implement the server….

- Create and initialize the ORB:

```
ORB orb = ORB.init(argv, null);
```

- Create the servant and register it with ORB

```
HelloServant helloRef = new HelloServant();
orb.connect(helloRef);
```

- Get the root NamingConext:

```
org.omg.CORBA.Object objRef = orb.resolve_initial_references(“NameService”);
NamingContext ncRef = NamingContectHelper.narrow(objRef);
```

- Bind the object reference in naming

```
NameComponent nc = new NameComponent("Hello", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, helloRef);
```

- Wait for invocations from clients:

```
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
sync.wait();
}
```

- Catch the exceptions

```
  } catch(Exception e) {
 System.err.println("ERROR: " + e);
  e.printStackTrace(System.out);
 } // end catch
} // end main()
} // end class
```

Step 5: write a client

- Import statements:

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
class HelloClient {
    public static void main(String argv[]) {
      try {
```

- Create and initialize the ORB:

```
ORB orb = ORB.init(argv, null);
```

- Create the root naming context:

```
org.omg.CORBA.Object objRef =
 orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

- Resolve the object reference in naming:

```
NameComponent nc = new NameComponent("Hello", " ");
NameComponent path[] = {nc};
Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
```

- Call the object:

```
String Hello = helloRef.sayHello();
System.out.println(Hello);
```

- Catch exception:

```
 } catch(Exception e) {
   System.out.println("ERROR : " + e);
 e.printStackTrace(System.out);
 } } } // end catch, main, class
```

Step 6: run the application

- Run the naming service:

```
prompt> tnameserver
```

- Run the server

```
prompt> java HelloServer
```

- Run the client

```
prompt> java HelloClient
Hello There
prompt>
```

CORBA Clients (details)

- Using the idlj compiler

- Initializing the ORB
- Helper classes
- Holder classes
- Exceptions
- The naming service

The idlj compiler

- Syntax: **idlj [options] idl-file**
- Examples:
  - **idlj myfile.idl** (generates client-side bindings)
  - Equivalent to: **idlj –fclient myfile.idl**
  - To generate client and server side bindings:
  - **idlj –fclient –fserver myfile.idl** OR
  - **idlj –fall myfile.idl**
- idlj supports other options….

The idlj compiler

- Syntax: **idlj [options] idl-file**
- Examples:
  - **idlj myfile.idl** (generates client-side bindings)
  - Equivalent to: **idlj –fclient myfile.idl**
  - To generate client and server side bindings:
  - **idlj –fclient –fserver myfile.idl** OR
  - **idlj –fall myfile.idl**
- idlj supports other options….

3.12 CORBA Event Service

1. Overview of CORBA Event Service
2. Communication Styles
3. Event Channel
4. Event Service Class Structure

The CORBA Event Service specification defines a model of communication that allows an application to send an event that will be received by any number of objects. The model provides two approaches to initiating event communication.

CORBA Model for Basic Client/Server Communications

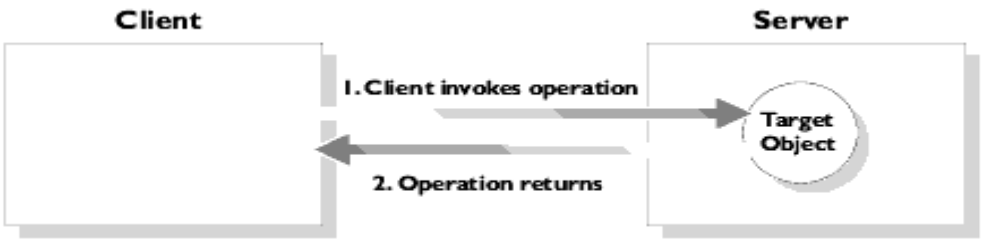


Figure 3.11.CORBA Model for Basic Client/Server Communications

In this model, a client application calls an IDL operation on a specified object in a server. The client waits for the call to complete and then receives confirmation of the return status. For any operation call there is a single client and a single server, and each must be available for the call to succeed. This simple, one-to-one communication model is fundamental to the CORBA architecture. However, some ORB applications need a more complex, indirect communication style. The CORBA Event Service defines a communication model that allows an application to send a message to objects in other applications without any knowledge about the objects that receive the message. The CORBA Event Service introduces the concept of *events* to CORBA communications.

An event originates at an event *supplier* and is transferred to any number of event *consumers*. Suppliers and consumers are completely decoupled: a supplier has no knowledge of the number of consumers or their identities, and consumers have no knowledge of which supplier generated a given event. In order to support this model, the CORBA Event Service introduces to CORBA a new architectural element, called an *event channel*.

An event channel mediates the transfer of events between the suppliers and consumers as follows:

- The event channel allows consumers to register interest in events, and stores this registration information.
- The channel accepts incoming events from suppliers.
- The channel forwards supplier-generated events to registered consumers.

Suppliers and consumers connect to the event channel and not directly to each other (Figure 1.2). From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers.

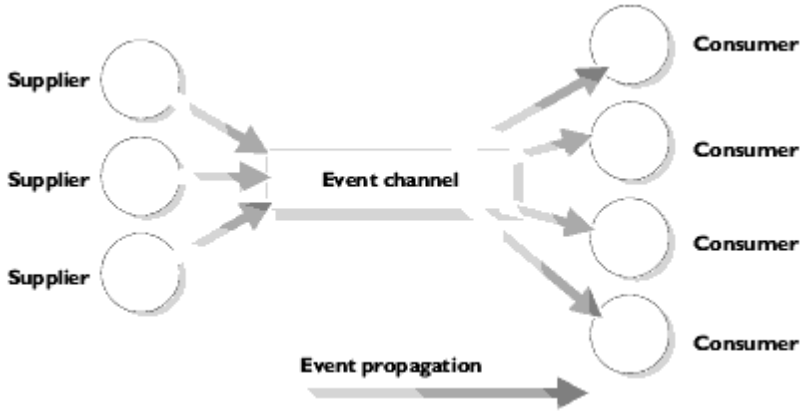


Figure 3.12: Suppliers and Consumers Communicating through an Event Channel

Any number of suppliers can issue events to any number of consumers using a single event channel. There is no correlation between the number of suppliers and the number of consumers, and new suppliers and consumers can be easily added to the system. In addition, any supplier or consumer can connect to more than one event channel.

Initiating Event Communication

CORBA specifies two approaches to initiating the transfer of events between suppliers and consumers. These approaches are called the *Push model* and the *Pull model*. In the Push model, suppliers initiate the transfer of events by sending those events to consumers. In the Pull model, consumers initiate the transfer of events by requesting those events from suppliers.

The Push Model

- In the Push model, a supplier generates events and actively passes them to a consumer.
- In this model, a consumer passively waits for events to arrive.
- suppliers in the Push model correspond to clients in normal CORBA applications, and consumers correspond to servers.

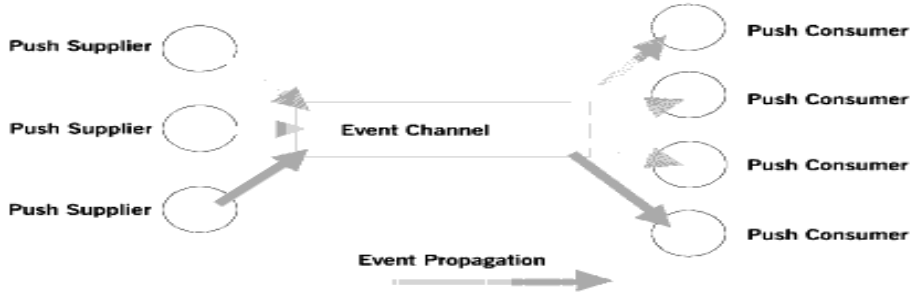


Figure 3.13 : Push model

In this architecture, a supplier initiates the transfer of an event by invoking an IDL operation on an object in the event channel. The event channel invokes a similar operation on an object in each consumer that has registered with the channel.

The Pull Model

- A consumer actively requests that a supplier generate an event.
- In this model, the supplier waits for a pull request to arrive. When a pull request arrives, event data is generated by the supplier and returned to the pulling consumer.
- Consumers in the Pull model correspond to clients in normal CORBA applications and suppliers correspond to servers.

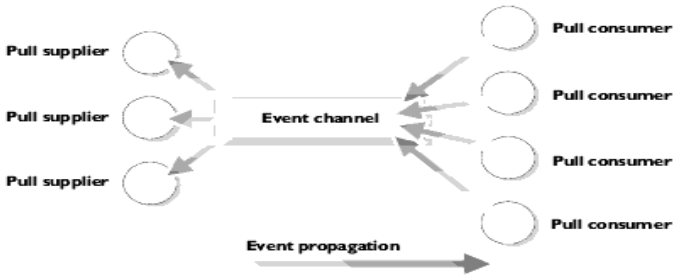


Figure 3.14 Pull Model

In this architecture, a consumer initiates the transfer of an event by invoking an IDL operation on an object in the event channel application. The event channel then invokes a similar operation on an object in each supplier. The event data is returned from the supplier to the event channel and then from the channel to the consumer which initiated the transfer.

Mixing the Push and Pull Models in a Single System

Suppliers and consumers are completely decoupled by an event channel, the Push and Pull models can be mixed in a single system. For example, suppliers may connect to an event channel using the Push model, while consumers connect using the Pull model

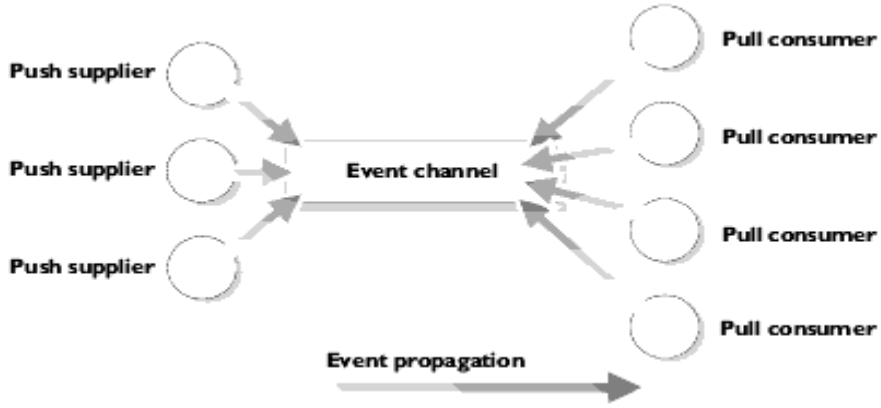


Figure 3.15 Push Model Suppliers and Pull Model Consumers in a Single System

In this case, both suppliers and consumers must participate in initiating event transfer. A supplier invokes an operation on an object in the event channel to transfer an event to the channel. A consumer then

invokes another operation on an event channel object to transfer the event data from the channel. The event channel stores events supplied by the push suppliers until some pull consumer requests an event, or until a push consumer connects to the event channel.

Types of Event Communication

The CORBA Event Service maps an event to a successfully completed sequence of operation calls. The operations and the sequence of calls are clearly defined for both Push and Pull models, and data about an event can be passed as operation parameters or return values. Event communication can take one of the two forms, *typed* or *untyped*.

Untyped Event Communication

In untyped event communication, an event is propagated by a series of generic push() or pull()operation calls. The push() operation takes a single parameter which stores the event data. The event data parameter is of type any, which allows any IDL defined data type to be passed between suppliers and consumers.

The pull() operation has no parameters but transmits event data in its return value, which is also of type any. Clearly, in both cases, the supplier and consumer applications must agree about the contents of the any parameter and return value if this data is to be useful.

Typed Event Communication

In typed event communication, a programmer defines application-specific IDL interfaces through which events are propagated. Rather than using push() and pull() operations and transmitting data using an any, a programmer defines an interface that suppliers and consumers use for the purpose of event communication. The operations defined on the interface may contain parameters defined in any suitable IDL data type. In the Push model, event communication is initiated simply by invoking operations defined on this interface.

The Pull model is more complex because event communication is initiated by invoking operations on an interface that is specially constructed from the application-specific interface that the programmer defines. Event communication is initiated by invoking operations on the constructed interface.

The Programming Interface to the Event Service

The CORBA Event Service specification defines the roles of consumer, supplier and event channel by describing IDL interfaces that each must support. The operations on these interfaces allow consumers and suppliers to register with an event channel to enable the propagation of events.

Module CosEvent Comm

The interfaces supporting the push and pull operation used to transmit event data are all defined in the CosEventComm module.These interface ise a notion of being connected while event data is being transmitted and disconnected when the party decide to stop receiving the data.

The Programming Interface for Untyped Events

The CORBA Event Service for untyped events defines interfaces for suppliers, consumers and event channels. It also defines a number of administration interfaces that allow suppliers and consumers to register with an event channel to allow the transfer of events between them.

Registration of Suppliers and Consumers with an Event Channel

A supplier connects to an event channel to indicate that it wishes to transfer events to consumers through that channel. A consumer connects to an event channel to register its interest in any events supplied through that channel. When a supplier or consumer no longer wishes to send or receive events, the application may disconnect itself from the event

The Push Model for Untyped Events

Four IDL interfaces support connection to and disconnection from event channels using the Push model:

- PushSupplier:
- PushConsumer
- ProxyPushConsumer
- ProxyPushSupplier

The interfaces PushSupplier and ProxyPushConsumer allow suppliers to supply events to an event channel. The interfaces PushConsumer and ProxyPushSupplier are specific to consumers, allowing them to receive events from an event channel.

These four interfaces are defined in IDL as follows:

```
// IDL
module CosEventComm {
  exception Disconnected {
  };

  interface PushConsumer {
    void push (in any data) raises (Disconnected);
    void disconnect_push_consumer ();
  };

  interface PushSupplier {
    void disconnect_push_supplier();
  };
};



module CosEventChannelAdmin {
  exception AlreadyConnected {
  };
```

```
  exception TypeError {
  };

  interface ProxyPushConsumer : CosEventComm::PushConsumer {
    void connect_push_supplier (
      in CosEventComm::PushSupplier push_supplier)
      raises (AlreadyConnected);
  };

  interface ProxyPushSupplier : CosEventComm::PushSupplier {
    void connect_push_consumer (
      in CosEventComm::PushConsumer push_consumer)
      raises (AlreadyConnected, TypeError);
  };

  ...
};
```

Connecting a Supplier

A supplier initiates connection to an event channel by obtaining a reference to an object of type ProxyPushConsumer in the channel. The supplier application may wish to be notified if the event channel terminates the connection. If so, the supplier then invokes the operation connect_push_supplier() on that object, passing a reference to an object of type PushSupplier as an operation parameter. If the ProxyPushConsumer is already connected to a PushSupplier, connect_push_supplier() will raise the exception AlreadyConnected.

Connecting a Consumer

A consumer first obtains a reference to a ProxyPushSupplier object implemented in the event channel. In order to register its interest in events from the channel, the consumer then invokes the operation connect_push_consumer() on the ProxyPushSupplier object. The consumer passes a reference to an object of type PushConsumer to the operation call. If ProxyPushSupplier is already connected to a PushConsumer, connect_push_consumer() will raise the exception AlreadyConnected.

Figure 3.16: Push Supplier and Push Consumer Connecting to an Event Channel in the Untyped Model

The Pull Model for Untyped Events

A similar set of IDL interfaces supports connection to and disconnection from event channels in the Pull model. These interfaces are:

- PullSupplier
- PullConsumer
- ProxyPullConsumer
- ProxyPullSupplier

The interfaces PullConsumer and ProxyPullSupplier allow consumers to request events from an event channel. The interfaces PullSupplier and ProxyPullConsumer allow an event channel to request events from suppliers. The Pull model interfaces are defined in IDL as follows:

```
// IDL
  module CosEventComm {
  exception Disconnected {
  };

  interface PullSupplier {
    any pull () raises (Disconnected);
    any try_pull (out boolean has_event) raises (Disconnected);
    void disconnect_pull_supplier();
  };

  interface PullConsumer {
    void disconnect_pull_consumer ();
  };
};


module CosEventChannelAdmin {
  exception AlreadyConnected {
```

```
  };

  exception TypeError {
  };

  interface ProxyPullSupplier : CosEventComm::PullSupplier {
    void connect_pull_consumer (
      in CosEventComm::PullConsumer pull_consumer)
      raises (AlreadyConnected);
  };

  interface ProxyPullConsumer : CosEventComm::PullConsumer {
    void connect_pull_supplier (
      in CosEventComm::PushSupplier pull_supplier)
      raises (AlreadyConnected, TypeError);
  };

  ...
};
```

Connecting a Consumer

In the Pull model, the transfer of events is initiated by consumers. A consumer initiates connection to an event channel by obtaining a reference to an object of type ProxyPullSupplier in the channel. The consumer application may wish to be notified if the event channel terminates the connection. If so, it invokes the operation connect_pull_consumer() on the ProxyPullSupplier object, passing a reference to an object of type PullConsumer as an operation parameter. If the ProxyPullSupplier is already connected to a PullConsumer, connect_pull_consumer() raises the exception AlreadyConnected.

Connecting a Supplier

To connect to an event channel, a pull supplier first obtains a reference to a ProxyPullConsumer object implemented in the event channel. The supplier then invokes the operation connect_pull_supplier() on the ProxyPullConsumer object, passing a reference to an object of type PullSupplier as the operation parameter. If the ProxyPullConsumer is already connected to a PullSupplier, connect_pull_supplier() raises the exception AlreadyConnected.

Figure 3.17: Pull Supplier and Pull Consumer Connecting to an Event Channel in the Untyped Model

Transfer of Untyped Events Through an Event Channel

The transfer of events from a supplier through an event channel to a consumer follows a simple pattern. Events originate at a supplier. In the Push model, a supplier pushes events into the event channel which in turn pushes the events to registered consumers. In the Pull model, consumers take the active role by requesting events from the event channel; the event channel, in turn, requests events from registered suppliers. Both methods of transfer are described for *untyped* events in the remainder of this section.

The Push Model

The supplier initiates event transfer by invoking the operation push() on a ProxyPushConsumer object in the event channel, passing the event data as a parameter of type any. The event channel then invokes a push() operation on the PushConsumer object in each registered consumer, again passing the event data as an operation parameter.

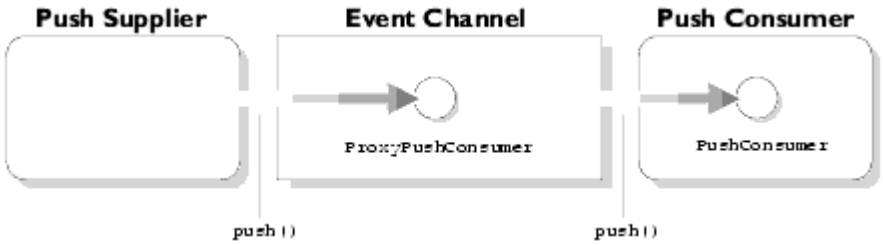


Figure 3.18: Transfer of an Event Through an Event Channel to a Consumer using the Untyped Push Model

Note that the supplier views the event channel as a single consumer and has no knowledge of the actual consumers. Likewise, the consumer views the event channel as a single supplier. In this way, the channel decouples the supplier and consumer.

The Pull Model

The consumer initiates event transfer in the Pull model. The consumer initiates event transfer in one of two ways as described below.

1. pull()

   The consumer invokes the pull() operation on a ProxyPullSupplier object in the event channel.

The event channel, if it does not already have an event, invokes a pull() operation on the PullSupplier object in each registered supplier. The pull() operation blocks until an event is available; the operation then returns the event data in its return value which is of type any. Thus, the consumer application blocks until the event channel can supply an event. The event channel, in turn, blocks until some supplier supplies an event to the channel.

2. try_pull()

   The consumer invokes the try_pull() operation on a ProxyPullSupplier object in the event channel.

   The event channel, in turn, invokes a try_pull() operation on the PullSupplier object in each registered supplier. If no supplier has an event available, try_pull() sets its boolean has_event parameter to false and returns immediately. If an event is available from some supplier, try_pull() sets the has_event parameter to true and returns the event data in its return value which is of type any.

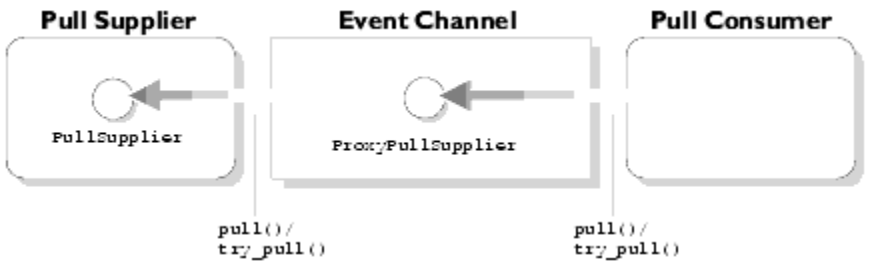


Figure 3.19: Transfer of an Event Through an Event Channel to a Consumer using the Untyped Pull Model

Note that, as in the Push model, the channel decouples suppliers and consumers. The consumer views the event channel as a single supplier and has no knowledge of the actual suppliers. Likewise, the supplier views the event channel as a single consumer.

Event Channel Administration Interfaces

The CORBA Event Service specification defines a set of interfaces that support event channel administration. The role of these interfaces is to allow a supplier or consumer to make initial contact with an event channel and to provide a set of standardized operations so that a supplier may obtain a ProxyPushConsumer or ProxyPullConsumer and a consumer may obtain a ProxyPushSupplier or ProxyPullSupplier object reference.

Each event channel supports the interface EventChannel, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
  ...

  interface EventChannel {
    ConsumerAdmin for_consumers ();
```

```
    SupplierAdmin for_suppliers ();
    void destroy ();
  };
};
```

If a supplier or consumer wishes to connect to an event channel, it must first obtain a reference to an EventChannel object in that channel. Typically, the event channel will publish a reference for this object, for example using the CORBA Naming Service.

A supplier then invokes the operation for_suppliers() on the EventChannel object. This operation returns a reference to an object of type SupplierAdmin, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer ();
    ProxyPullConsumer obtain_pull_consumer ();
  };

  ...
};
```

To obtain a reference to a ProxyPushConsumer object in the event channel, the supplier invokes the operation obtain_push_consumer() on the SupplierAdmin object. At this point, the supplier is ready to connect to the channel and begin transferring events using the Push model.

The supplier invokes the operation obtain_pull_consumer() on the SupplierAdmin object if it wishes to obtain a ProxyPullConsumer. The supplier is then ready to connect to the channel and to transfer events using the Pull model.

Similarly, a consumer invokes the operation for_consumers() on an EventChannel object in order to obtain a reference to an object of type ConsumerAdmin, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier ();
    ProxyPullSupplier obtain_pull_supplier ();
  };

  ...
};
```

If the consumer is using the Push model, it then invokes the operation obtain_push_supplier() to obtain a reference to a ProxyPushSupplier. If the consumer is using the Pull model, it invokes the operation

obtain_pull_supplier() to obtain a reference to a ProxyPullSupplier object in the event channel. The consumer is then free to register its interest in events propagated through the channel.

3.13 Practical applications

To write a program to develop a middle-ware component for checkin the given number is prime or not.

- Creating a module which contains a set of interface program and save it as a Prime.idl makes sure that module and interface name is different.
- Next declare the implementation file with ImplBase concept & contains details about find Prime number.
- Create a server program that initialize the ORB name the server object & bind the server object with the implementation file now server is ready.
- Create a client program by initialize the ORB. Make sure the name of server object and client object declared resembles the frame.
- Then provide the input to the client program and obtain the suitable output corresponding to it.

Creating a module

```
//Prime.idl
module Prime
{
 interface PrimeInt
 {
  string checkPrime(in long n);
 };
};
```

Generate Java classes

- Command Line tool

  idlj -fall -oldImplBase Prime.idl

f→Controls whether to generate client/server side mapping.ie which flavor of mapping to generate.

all → Generating both server and client mapping.

- Use *idlj* on CORBA IDL interface and generates the following items
  - The equivalent Java interface: PrimeInt.java
  - The Portable Object Adapter (POA) abstract class PrimePOA.java (since J2SE 1.4) for Servant class to extend
  - The proxy class for client stub, *_PrimeIntStub.java*
  - Classes called helpers and holders, one for each of the types defined in the IDL interface

- Helper contains the *narrow* method, which is used to cast down from a given object reference to the class to which it belongs
- Holder deals with *out* and *inout* arguments, which cannot be mapped directly in Java

– Java classes corresponding to each of the *struct*s defined within the IDL interface

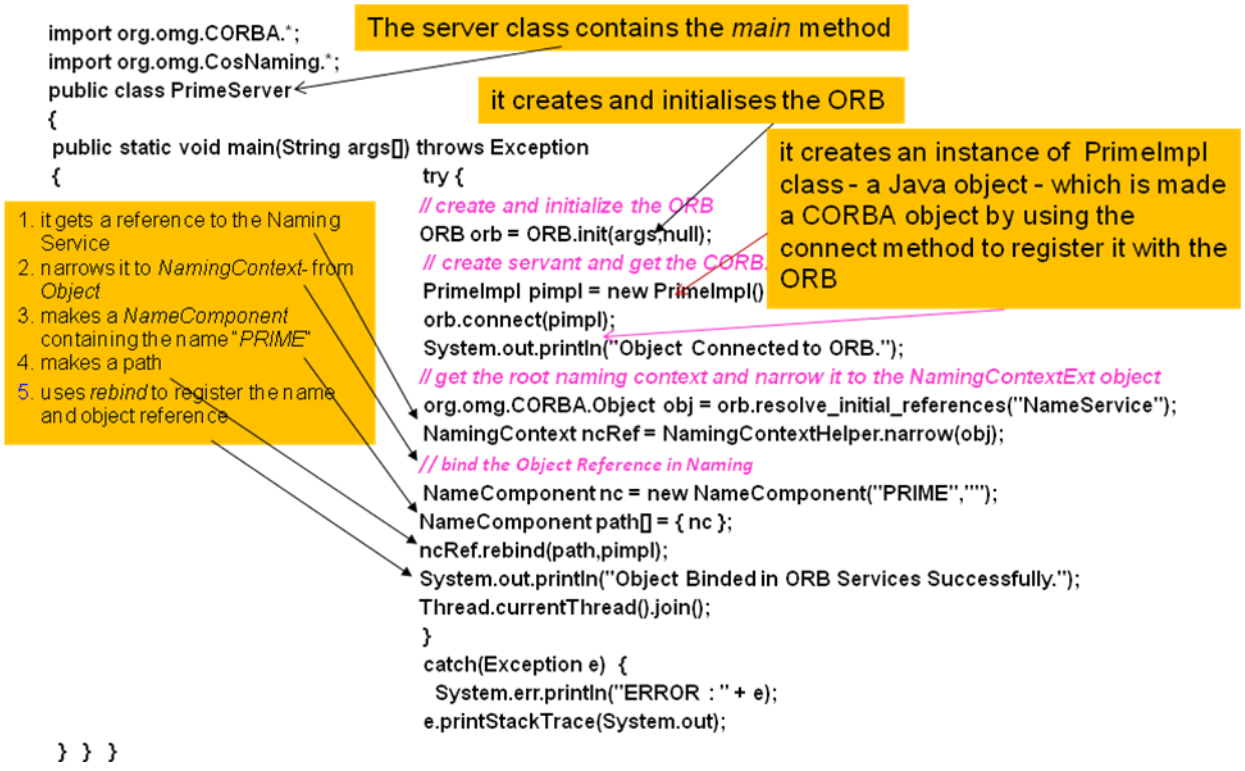
Compile & Execution Steps:

- idlj -fall -oldImplBase Prime.idl
- javac *.java (compile implementation, server, client programs.)
- start orbd -ORBInitialPort 1050
- start java PrimeServer -ORBInitialPort 1050
- java PrimeClient -ORBInitialPort 1050

Implement the Servant

```
import Prime.*;
public class PrimeImpl extends _PrimeIntImplBase
{
 public String checkPrime(int n)
 {
  int count = 0,flag = 0;
  int num = n;
  String result;
  for (int i=2;i<(num/2);i++)
  {
   if  ((num % i) == 0)
        flag = 1;
  }
  if (flag == 0)
     result = "The Given Number is a Prime Number.";
  else
     result = "The Given Number is not a Prime Number.";
  return result;
 }
}
```

Implement the server



Commands explanation

- *activate*: make the object enabled in CORBA
- *servant_to_reference:* get the object reference from the servant class
- *resolve_initial_references*: first lookup of POA
- *narrow*: cast CORBA object reference to the preferred class
- *to_name:* convert between string value and name component path
- *rebind*: bind and rebind the object reference to the naming service

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import Prime.*;
public class PrimeClient
{
 public static void main(String args[]) throws Exception
 {
  try
  {
// create and initialize the ORB
   ORB orb = ORB.init(args,null);
 // get the root naming context
   org.omg.CORBA.Object obj = orb.resolve_initial_references("NameService");
 // Use NamingContextExt instead of NamingContext, part of the Interoperable naming
     Service.
   NamingContext ncRef = NamingContextHelper.narrow(obj);
   NameComponent nc = new NameComponent("PRIME","");
   NameComponent path[] = { nc };
 // resolve the Object Reference in Naming
   PrimeInt SSK = PrimeIntHelper.narrow(ncRef.resolve(path));
  java.io.DataInputStream d = new java.io.DataInputStream(System.in);
  System.out.print("\n\t\t\tEnter the Number to Check : ");
  int n = Integer.parseInt(d.readLine());
  String result = SSK.checkPrime(n);
  System.out.println("\n\t\t\t" + result);
 }
 catch(Exception e)
 {
  System.err.println("ERROR : " + e);
  e.printStackTrace(System.out);
 }
}
}
```

1. it contacts the *NamingService* for initial context
2. Narrows it to *NamingContext*
3. It makes a name component
4. It makes a path
5. It gets a reference to the CORBA object called "PrimeInt", using *resolve* and narrows it

- Run the Object Request Broker Daemon (usedby clients for look up and object invocation on servers)
  - start orbd -ORBInitialPort 1050
- Run the server

  start java PrimeServer -ORBInitialPort 1050
- Run the client
  - java PrimeClient -ORBInitialPort 1050

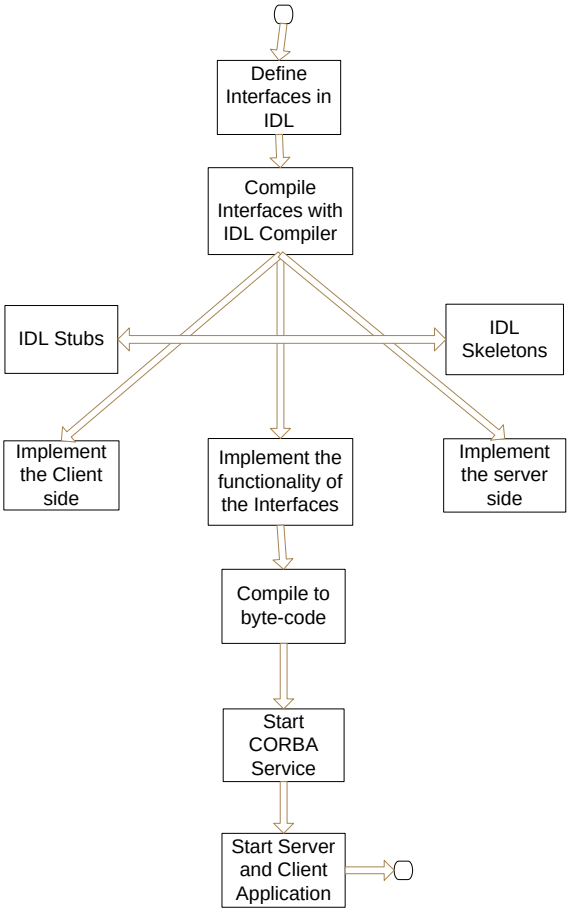


Figure 3.20 : CORBA implementation workflow

UNIT IV COM OVERVIEW 10 hrs.

COM – Distributed COM – COM facilities and services – Applying COM objects – COM interface – Query interface – Reference counting –Dynamic linking – Class factory – Component issue.

4.1 COM

- COM components consists of executable code distributed either as Win 32 dynamic link libraries(DLL) or as executables (EXEs)
- COM components can be shipped in binary form(providing a standard for components to follow)
- COM components are fully language independent. The COM components can be developed using any procedural language(eg., c, Java.,) and any language can be modified to use COM components (eg., Visual Basic (Defining language independent architecture)
- COM component can be upgraded without breaking old clients (allowing for multiple versions of components)
- COM components can be transparently relocated on a network. The component on remote system is treated same as component on the local system (supporting transparent links to remote components)
- COM has API, COM library which provides component management services that are useful for all clients and components

Note:

The features include the *separation of interfaces and implementations* support for *objects with*

- *multiple interfaces*
- *language neutrality*
- *run-time binary software reuse*
- *location transparency*
- architecture for *extensibility*
- support for *indirection*
- approach to *versioning* and
- different styles of *server lifetime management*
    - Object instances are created when clients make such requests
    - Reference counting is used to manage server lifetime: the server increments the count upon returning an interface pointer. Each client must follow a set of rules to increment the count (when duplicating an interface pointer, for

example) and to decrement the count when finish using a pointer. When the count drops to zero, the server object realizes that it is no longer serving any client and can therefore be destroyed

COM Architecture

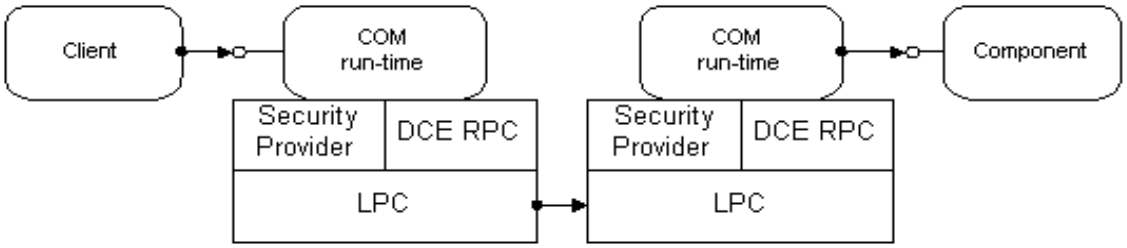


Figure 4.1 Communication details handled by the COM run-time

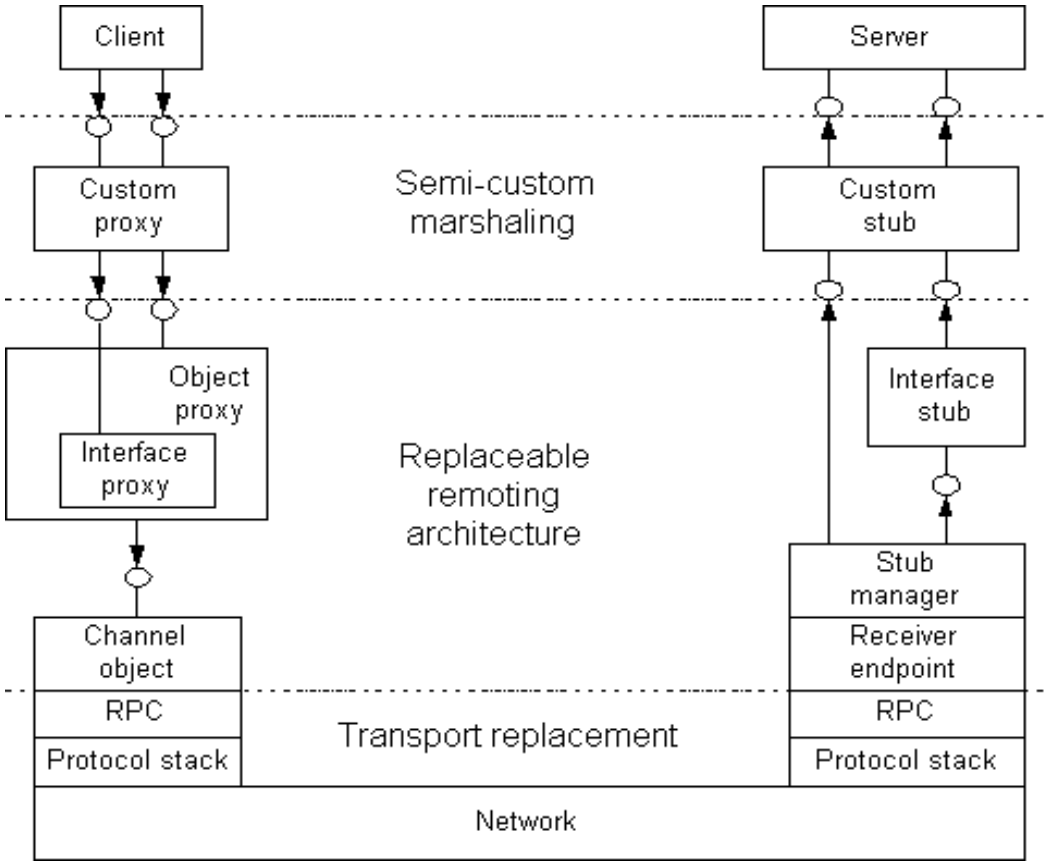


Figure 4.2. COM remoting architecture and extensibility

Remoting architecture : The standard remoting architecture includes

1) *object proxies* that act as the client-side representatives of server objects and connect directly to the client.
2) *interface proxies* that perform client-side data marshaling and are aggregated into object proxies.
3) client-side *channel objects* that use remote procedure calls (RPCs) to forward marshaled calls.
4) server-side endpoints that receive RPC requests.
5) server-side *stub manager* that dispatches calls to appropriate interface stubs.
6) *interface stubs* that perform server-side data marshaling and make actual calls on the objects and
7) *standard marshaler* that marshals interface pointers into *object references* on the server side and unmarshals the object references on the client side. Note that interface proxies and stubs are application-specific and are generated by running an Interface Definition Language (IDL) compiler on application-supplied IDL files. The other objects are application-independent and are provided by COM.

Language Neutrality

Various languages can be used to create components.

- VB, Delphi for rapid development
- VC++, Java for advanced development
- Micro Focus COBOL
- Even more languages can be used to use COM components
- Additionally scripting languages like VB Script, Jscript

Location Transparency

- COM Object locations are stored in registry
- Applications make calls using the globally unique CLSID
- Path to COM server, or remote computer to run DCOM server is not needed by the application

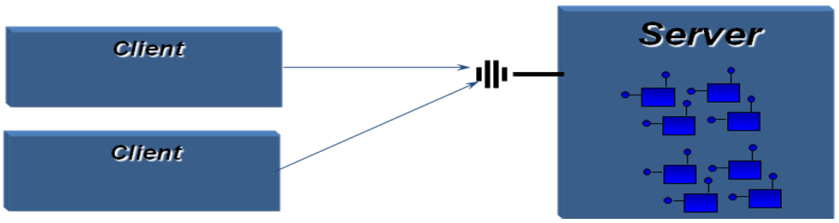
Connection Management

- Low Bandwidth
  - Header is 28 bytes over DCE-RPC
  - Keep-Alive Messages bundled for all connections between Machines
  - COM employs an efficient pinging protocol to detect if clients are active

- COM uses reference counting mechanism to do garbage collection

Efficient and Scalable

- Multiplexing - Single Port per-protocol, per server process, regardless of # of objects
- Scalable - Connection-Less Protocols like UDP Preferred
- Established Connection-Oriented (TCP) Sessions Reused by same client



Load Balancing

- DCOM does not transparently provide load balancing
- Makes it easy to implement load balancing
  - Static load balancing
  - Dynamic load balancing by means of a dedicated referral component

Technical Significance of COM

- COM has had a fundamental impact on software development processes for many engineering organizations
  - Integration is significantly simpler with standardized interfaces
  - Provides effective mechanism for software reuse that complements object oriented design methods as supported by C++
  - Microsoft developed COM into a scalable technology
  - However, COM is a mature technology and Microsoft is now turning, for a variety of good reasons, to its new .Net technologies

Future of COM

- All of the Microsoft technologies have been COM centered.
  - Windows is full of COM components, Office and Visio are implemented in COM.
- COM has competitors. CORBA and JAVI RMI/beans provide very similar services.

- Microsoft has shifted a lot of its focus to the .Net platform that co-exists with COM but is not COM centered.
  - Microsoft's enterprise platform is moving away from COM to .Net
- CORBA and Java seem to be moving toward niche markets.
  - CORBA headed toward sharing the enterprise market with JavaEE and .Net?
  - Java has not implemented desktop services nearly as quickly as COM and has some significant performance disadvantages.
  - Java's strongest arena is the internet, but .Net will give it strong competition with its Web Services.
  - .Net, not COM, will probably be CORBA and Java's biggest competitor.

USE OF COM

Component Object Model (COM) is a binary-interface standard for software componentry introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages. The term *COM* is often used in the Microsoft software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.

4.2 DCOM

- Called as COM with a long wire
- Combining components those are located on different machines (DCOM extends COM to support communication among objects on different computers)
- DCOM protocol enables software components to communicate directly over a network
- DCOM is Microsoft product, an open standard and has been ported to other platforms (on several UNIX platforms)
- Windows NT 4.0 is the first Microsoft OS to support DCOM (developed by Microsoft previously called Network OLE)
- Designed for use across multiple network transports including Internet protocols such as HTTP

Distributed Component Object Model

- Designed by Microsoft
- Based on Component Object Model (COM)
- Addresses issues such as:
  - Interoperability
    - Different applications, platforms, languages
  - Versioning
    - Compatibility between a new version of a server and old versions of clients
      - New interfaces should preserve the old interface
  - Naming
    - Use Globally unique identifiers

- Client side infrastructure in DCOM is called a proxy
- Server side is called as stub
- Comparing with RMI
- Client side infrastructure called as stub

DCOM Architecture

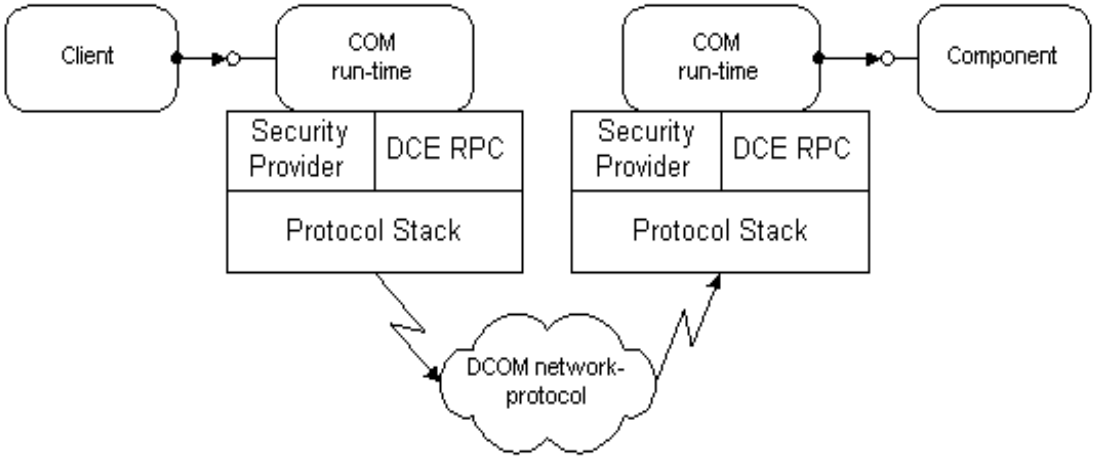
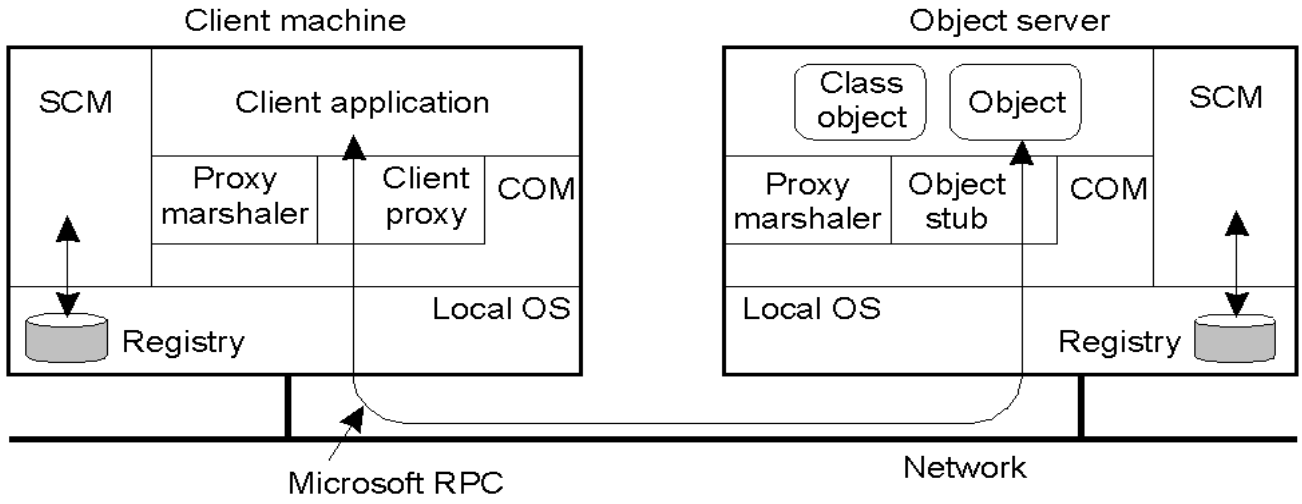


Figure 4.3 DCOM architecture



- SCM: Service Control Manager

Explanation:

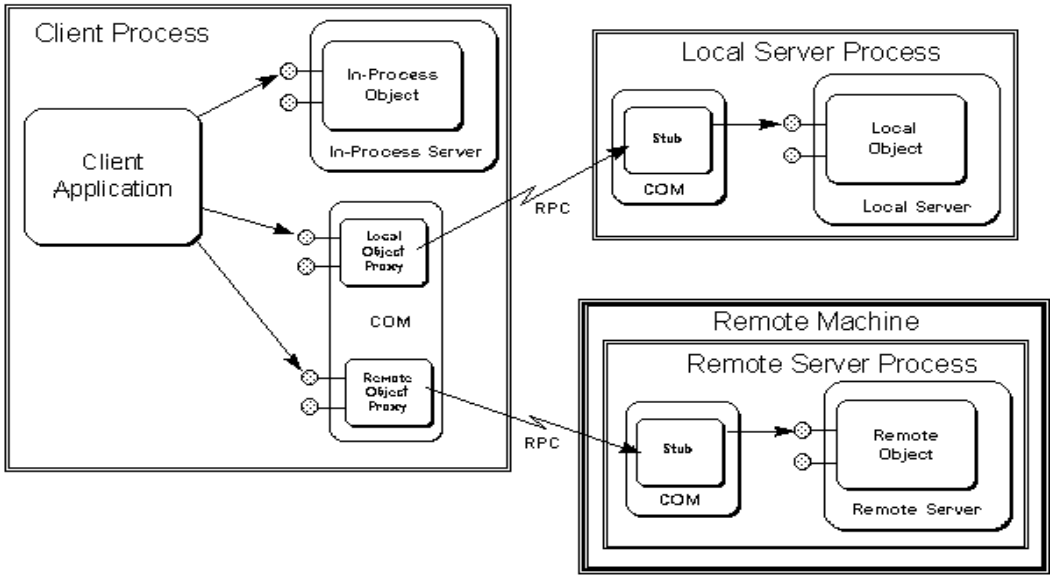
- COM uses a mechanism to pass parameters and return values accross process boundaries.
- The client calls the functions of the interface proxy, but the application need not worry about the details of the process. The proxy has exactly the same functions as the target interface.
- The proxy marshalls the data and sends to the server side.
- The stub gets the data sent from the client proxy and unmarshalls the data. The parameters are extracted and passed to the actual interface. The server implementation feels that the request comes from a real client.
- The stub calls the function on the server. But the server need not have to worry about these details. The server processes the parameters and returns the parameters to the stub.
- The stub marshalls the return values and send to the client proxy.
- The proxy unmarshalls the return values and passed to the client.

When an in-process object is involved, COM can simply pass the pointer directly from the object to the client, because that pointer is valid in the client's address space. Calls through that pointer end up directly in the object code, as they should, making the in-process case a fast calling model—just as fast as using raw DLLs

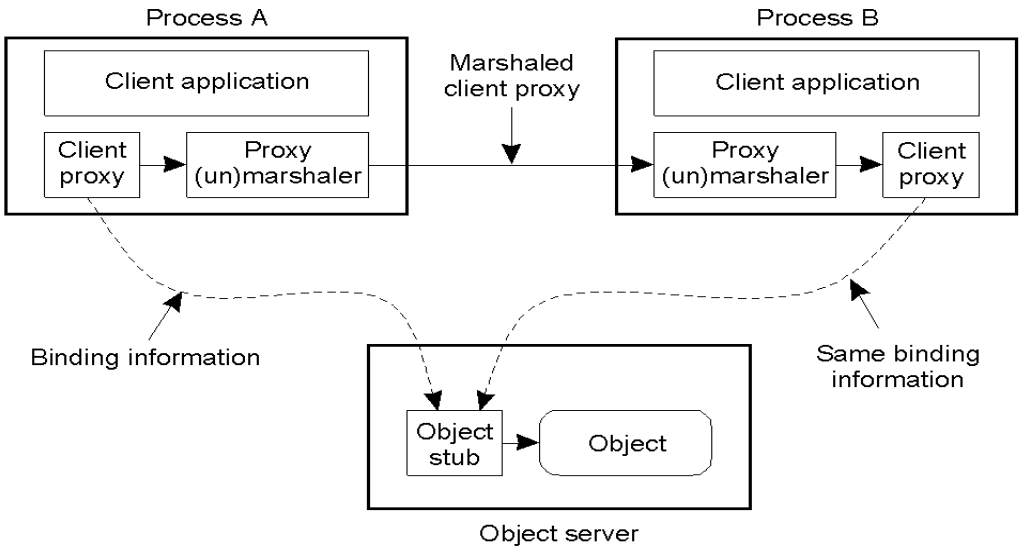
This "marshalling" sequence creates a "proxy" object and a "stub" object that handle the cross-process communication details for that interface. COM creates the "stub"

in the object's process and has the stub manage the real interface pointer. COM then creates the "proxy" in the client's process, and connects it to the stub. The proxy then supplies the interface pointer that is given to the client.

Users may choose to implement their own custom marshalling by implementing the IMarshall interface. The COM runtime first queried the IMarshall interface. If it cannot find one, it uses the universal type library marshalling.



Passing an Object Reference in DCOM
(with custom marshaling)



DCOM Properties

- Distributed shared memory management
  - DCOM provides interfaces for distributed components to share memory
- Network interoperability and transparency
- Dynamic loading and unloading
  - DCOM manages reference counts to objects
  - Unloads objects whose reference count is 0
- Status reporting
  - Of remote execution using HRESULT struct

DCOM Services

- DCOM is responsible for initializing a connection between components, and
  - Negotiating protocols for communication
- DCOM provides support for persistent storage
  - Objects can persist

Platform Neutrality

- DCOM run-time is available for various platforms
  - Win32 platforms, Solaris, DEC UNIX, HPUX, Linux, MVS, VMS, Mac
  - Cross-Platform Interoperability Standard
  - Per-Platform binary standard
  - Unlike java, DCOM can utilize powerful platform-specific services and optimizations
  - Less abstraction layers prevents additional overheads

Benefits of DCOM

- Large User Base and Component Market
- Binary Software Integration
  - Large-scale software reuse (no source code)
  - Cross-language software reuse.
- On-line software update.
  - Allows updating a component in an application without recompilation, relinking or even restarting.
  - Multiple interfaces per object
- Wide selection of programming tools available, most of which provide automation of standardcode.

Microsoft Transaction Server (MTS)

In two-tier client-server systems management of resources at the server application developer's responsibility. This also makes interaction with legacy database servers, which are not integrated with Windows NT well, very difficult.

Hence, Microsoft introduced the three-tier architecture, which consists of the usual client and database server tiers, but adds a middle tier, which handles the server object creation, deletion and requests, rather erroneously named Microsoft Transaction Server (MTS), because it handles much more than just transactions [16].

MTS is a COM component hosting environment that

- Manages system resources (e.g. processes, threads, and connections)
- Manages server object creation, execution and deletion
- Automatically initiates and controls transactions
- Implements security so that unauthorized users cannot access the application
- Provides tools for configuring, managing, and deploying the application's components.

The key benefit of this is that it frees developers from many of these aspects of middle tier application development and allows them to concentrate on the business logic that they are implementing. It forms the middle tier of a distributed system and hosts the COM components that do the work. In a typical system, it uses Open Database Connectivity (ODBC) or OLE DB to connect to the database server.

A very useful functionality that MTS provides is Just in Time Activation of components, which drastically reduces the number of components and database connections, when used along with Object Pooling. MTS doesn't really create a component, when the client initiates it. It waits until the client actually invokes one of the methods. MTS usually destroys the component when it has finished a method call for a client.

MTS components are dynamically created and destroyed each time they are required – Just in Time Activation. Any resources that they use are pooled and shared between the components as they run.

MTS offers additional utilities (for example, Shared Property Manager) that help the developer in maintaining the state of the object. The other problem is that if the object themselves are small but store a lot of state for each client, recycling and pooling will actually decrease performance. Recycling and Pooling would help only if the object takes long time to create and holds up a lot of system resources. The decision of whether to support recycling should be based on the expense of creating new objects vs. the cost of holding up the resources of that object while it is in the pool.

COM+

COM and MTS were introduced with Windows NT. From Windows 2000, the same services are provided by COM+, which is a combination of COM and MTS and it offers some new services that didn't exist in COM or MTS. COM+ doesn't replace COM/MTS, it extends it. It is backward compatible, so migrating MTS packages is simple. The improvements include a new threading model, and better way to handle database connections.

COM+ also improves on COM by providing default implementations of standard COM interfaces. This relieves the programmers of implementing obscure interfaces like IConnectionPoint, IconnectionPointContainer etc. It also automates writing housekeeping code, which takes up as much as 30 percent of the time for COM developers.

COM is easier to program in Java and VB, than C++. Although COM claims to be a language independent binary standard for components, there are various internal implementation issues that has resulted in peculiarities regarding the way it uses and builds COM based components. COM+ aims to smoothen outthese differences and make a component written in any language automatically accessible to all other languages supporting COM+.

COM OBJECTS AND INTERFACE

Interface:

- collection of functions by which applications interact with each other and with the system
- strongly typed contract between software components to provide a small but useful set of semantically related operations (methods) : Component Object Design
- definition of an expected behavior and expected responsabilities

COM Interface:

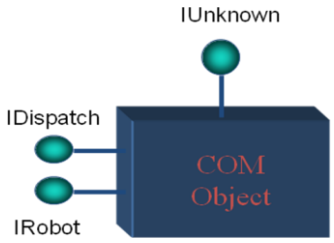
- COM object appears in memory much like a C++ object
- Unlike C++ objects, however, a client never has direct access to the COM object in its entirety. Instead, clients always access the object through clearly defined contracts: the interfaces that the object supports, *and only those interfaces*
- interface is a strongly-typed group of semantically-related functions, also called "interface member functions“

The name of an interface is always prefixed with an "I" by convention, as in IUnknown.

- Client can communicate with the COM component only through an interface
- Components should be able to add and remove interface without breaking existing users
- COM clients only interact with pointers to interfaces
- COM components can implement multiple interfaces
- Interfaces are strongly typed
- Interfaces are immutable
- An interface is not a class
- remember : considered logically immutable
- reason : removing the potential for version incompatibility
- new functionality : can be exposed through a different interface
- is not a component object denotes behavior only,
- not state component objects can implement multiple interfaces
- is strongly typed

COM Design Principles..

- Encapsulation
  - Black box - no leakage of implementation details



- All object manipulation through strict interfaces

- Polymorphism
  - via multiple interfaces per class

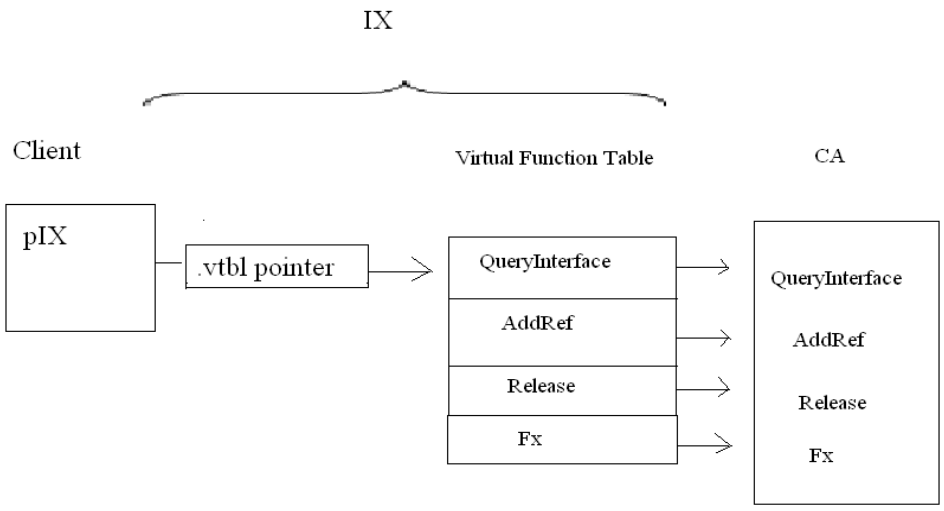
    “Discoverable”: QueryInterface

  - COM Binary Standard

    A standard way to : lay out virtual function tables (vtables) in memory call functions through the vtables

    any language that can call functions via pointers (C, C++, Small Talk, Ada, etc.,) can be used to write components that can interoperate with other components written into the same binary standard

COM interfaces inherit from Iunknown and QueryInterface, AddRef and Release as the first three entries in their vtbls



Component Object Library

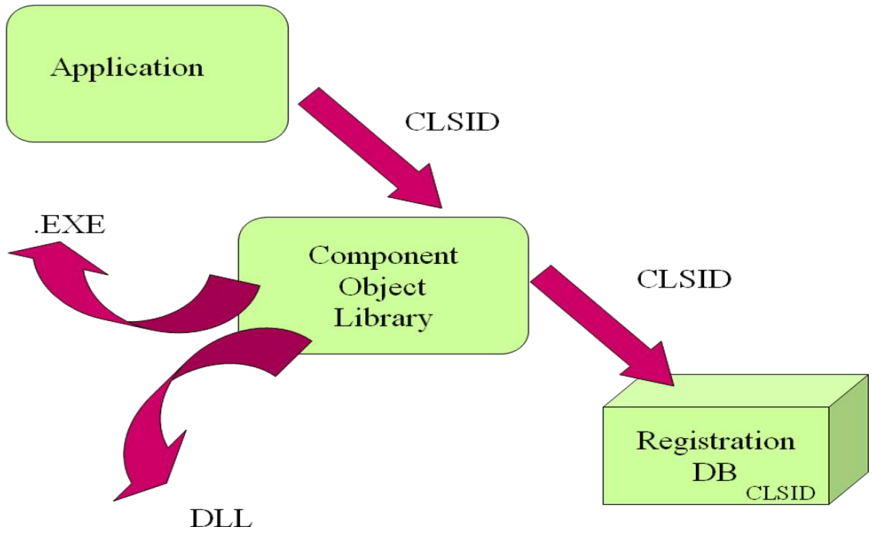
A system component that provides the mechanics of COM:

- provides the ability to make IUnknown calls across processes

- encapsulates all the “legwork” associated with launching components and establishing connections between components

GUID’s (or UUID’s)

- Globally Unique Identifiers (Universally Unique Identifiers)
- Needed to avoid name collisions
- A class is associated with a GUID (CLSID)
- An interface is associated with a GUID (IID)
- The Windows Registry: a hierarchical database.



COM Architecture A scalable programming model

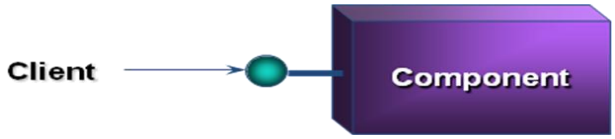
COM Servers

Supports three types of servers to implement components

- In – Process server
- Local server
- Remote server

In- Process server…

- In the same process

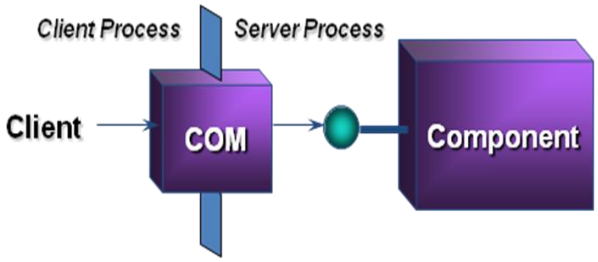


- Fast, direct function calls

- Implemented as DLL
- Executes with same process space as the application
- Performance overhead of invoking the server is less

Local Server

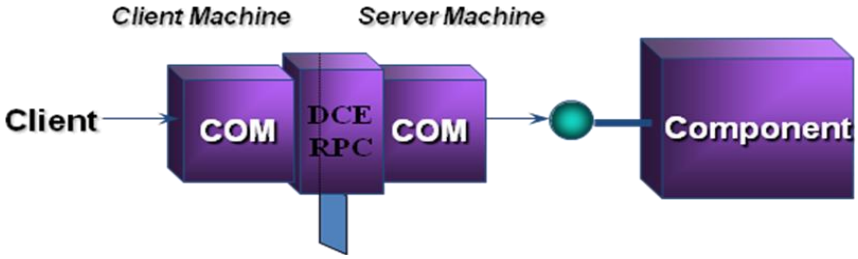
- On the same machine - Fast, secure IPC



  - Executes as a separate process on the same computer
  - COM runtime allows communication between an application and the server (uses high speed interprocess communication protocol)

Remote Server

- Across machines
  - Secure, reliable and flexible
    DCE-RPC based *DCOM* protocol



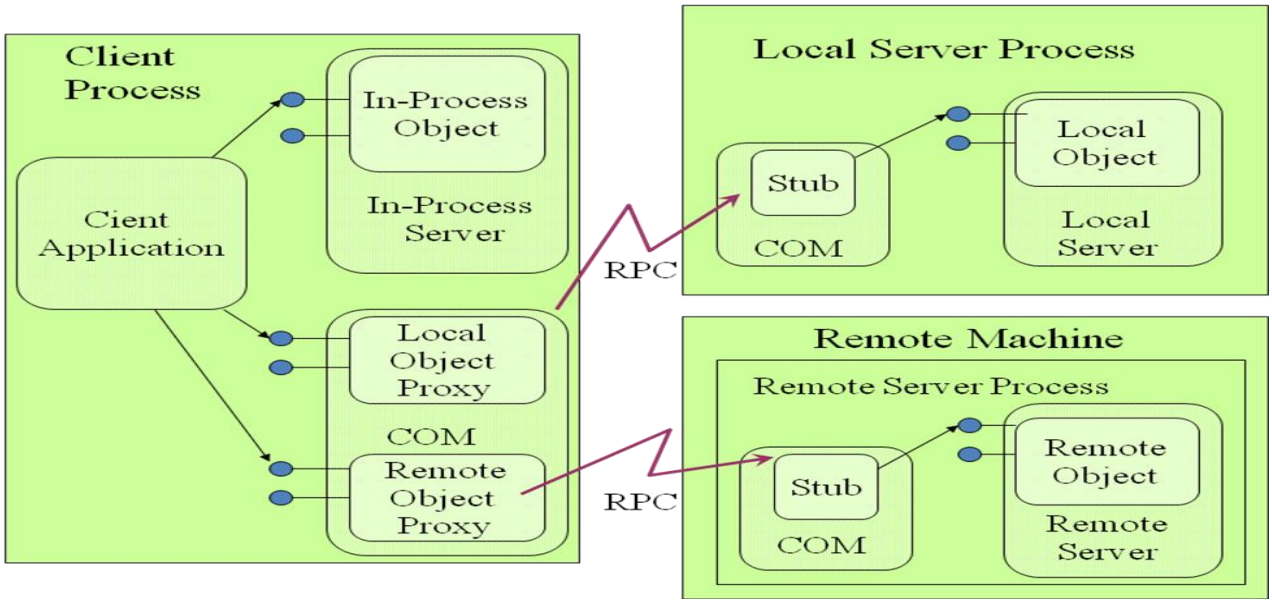
  - Execute on a remote computer
  - DCOM extends COM and provides RPC based infrastructure for communication

COM server

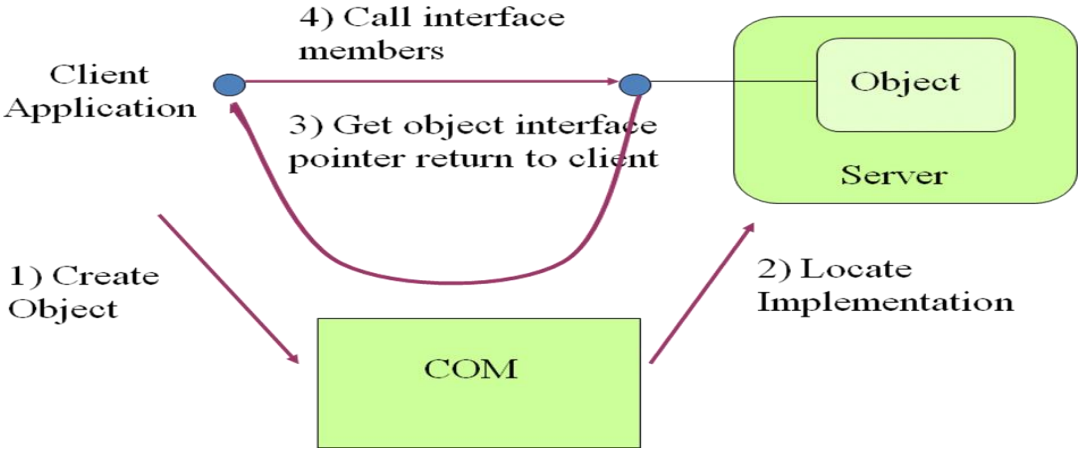
In general :

some piece of code that implements some component object such that the Component Object Library and its services can run that code and have it create component objects. Three ways in which a client can access COM objects provided by a server:

- In-Process Server
- Local Object Proxy
- Remote Object Proxy



COM object: What happens when a client wishes to create and use a COM object?



When an application uses a COM object:

1) Initializes the system: CoInitialize
2) Calls CoCreateInstance (exported by COMPOBJ.DLL) passing the CLSID of the used object
   - uses the registry to discover which server implements the desidered class
   - ask the server to create an instance

- receives from the server a pointer to the IUknown interface

3) uses IUnknown . QueryInterface to access the wanted interface
4) uses Addref and Release to manage the object life cycle
5) in the end calls CoUninitialize

COM SERVER AND CLIENT SIDE

Server Side

One of the cardinal rules of COM is that a COM object can only be accessed through an interface. The client program is completely isolated from the server's implementation through interfaces. The client program knows nothing about the COM object or C++ class that implements the COM object. All it can see is the interface. Every COM object supports at least one interface called IUnknown. All custom interfaces inherit from this interface. The IUnknown interface comes with three methods, QueryInterface, AddRef and Release. Through IUnknown, it can control the lifetime of an object and invoke QueryInterface. QueryInterface is the basic function in COM through which one piece of software determines what other interfaces are supported by a component. Call to Release results in decrementing the reference count. Reference counting provides a lifetime control mechanism that allows a COM object to keep track of its clients and can delete itself when it is no longer needed. The interface consists of virtual declarations of member functions implemented by the server. Each interface has a globally unique identifier (GUID) called the IID. Similarly, each COM object class is assigned a unique class ID (CLSID). The identifier is 128 bit long hexadecimal number that is unqiue in the universe. These unique indentifiers (for both CLSID and IID) are entered in the system registry by the server.

On the Server side, the interface and shared objects are identified along with their GUIDs in the IDL file. The IDL language used by COM/ DCOM is called *MIDL.* Interface specifications are compiled by the standard *Microsoft IDL Compiler* (also denoted as *MIDL),* which creates the code of sever stubs and client proxies. However, the code of stubs and proxies can be generated by other Microsoft compilers as well (e. g., by Visual C++).

In DCOM , we assume server imple ments a CTest object that exposes an ITest interface.

The interface exposes one method MethodA(). Typical interface declaration looks like the following. Note the unique GUID for the interface.

```
uuid(37483ED1-517E-412D-A715-0737F151EE65),

interface ITest : IUnknown
{
HRESULT MethodA([in]int* param, [out]int* retval);
};
```

DCOM requires that all methods return a 32-bit error code called an HRESULT . At the language/tool level, a set of conventions and system provided services (called the IErrorInfo object) allows failure HRESULTs to be converted into exceptions in a way natural to thelanguage. For example, in Microsoft Visual C++ 6.0, client programmers can use standard C++ try/catch blocks to catch errors from COM method invocations; the compiler generates the correct code to map the failure HRESULT into a correct usage of IErrorInfo, effectively translating the failure return code into an exception. Similarly, tools can allow programmers to "throwexceptions" instead of returning failure codes.

Client Side

On the client side, the implementation varies slightly depending on the type of server (InProc or EXE). But the basic steps are the following. Continuing on from our earlier example of a Ctest object on the server exposing an ITest interface.

```
ITest *pi // pointer to xxx COM interface
CoInitialize() // Initialize COM
CoCreateInstance(,,,,&pi) // create interface
pi->MethodA(); // call method
pi->Release(); // free interface
Couninitialize(); // Uninit COM
```

CoInitialize()initializes the COM interface.

Alternatively, OleInitialize() may be called, though it calls CoInitialize() internally. After initializing COM, the client instantiates the desired COM class containing the

implementation of the interface, with CoCreateInstance. CoCreateInstance() takes the CLSID_CTest and IID_ITest for the interface as its arguments. It can thought to be equivalent of C++ new operator. CoCreateInstance uses the CLSID to search for a match in the HKEY_CLASSES_ROOT\CLSID section of the registry in order to locate the desired component. The third parameter is one of CLSCTX_INPROC_SERVER, CLSCTX_LOCAL_SERVER, CLSCTX_REMOTE_SERVER, specifying whether the component may be available as an in-proc version, or as a local or remote version. The fourth parameter is the Interface identifier or the IID. Then the client accesses the methods exposed by the interface. After it is done using the object, it calls the Release method of the interface (always provided by IUnknown), which results in
decrementing the reference count on the server. Finally, CoUninitialize() is called to perform the cleanup duties. It closes the COM library, freeing any resources it maintains and forcing all RPC connections to close.

There are various versions for each of the functions mentioned above, each of which can be used in special cases. For instance, alternatives to CoCreateInstance() are CoCreateInstanceEx (used to run executable component across network using DCOM), CoGetInstanceFromFile (creates a new instance and initializes it from a file), CoGetClassObject (returns an interface pointer to a "class factory object" that can be used to create one or more uninitialized instances of the object class CLSID) etc.

4.5 COM Interface

COM Configuration Management

- COM objects and their interfaces are identified by Globally Unique Identifiers (GUIDs). These are 128 bit numbers generated by an algorithm based on machine identity and date
- COM requires that interfaces are immutable. That is, once an interface is published it will never change. A component may change its implementation, removing latent errors or improving performance, but interface syntax and semantics must be fixed
- Components may add new functionality, expressed by additional interfaces, but the component must continue to support its original interface set

COM Interface Policy

- COM components must implement the standard IUnknown interface, and thus all COM interfaces are derived from Iunknown

- All COM interfaces must declare the IUnknown methods, usually done by inheriting from IUnknown.

- All COM objects are required to implement the IUnknown interface along with their own operations.
- IUnknown interface consists of three methods:

1) QueryInterface( )

- QueryInterface, used by clients to inquire if a component supports other specific interfaces

- QueryInterface allows clients to dynamically discover whether or not an interface is supported by a component object; if so, it returns an interface pointer from a component object

- which by specifying an IID allows a caller to retrieve references to the different interfaces the component implements. The effect of QueryInterface() is similar to dynamic_cast<> in C++ or casts in Java and C#

2) AddRef() and Release(), which implement reference counting and controls the lifetime of interfaces

- AddRef, used by COM to increment a reference count maintained by all COM objects
- Release, used by clients to decrement the reference count when finished with interface. When the reference count decrements to zero the object's server is unloaded from memory.

- A COM component's interfaces are required to exhibit the reflexive, symmetric, and transitive properties

  - reflexive property refers to the ability for the QueryInterface() call on a given interface with the interface's ID to return the same instance of the interface

  - symmetric property requires that when interface B is retrieved from interface A via QueryInterface(), interface A is retrievable from interface B as well

- transitive property requires that if interface B is obtainable from interface A and interface C is obtainable from interface B, then interface C should be retrievable from interface A

- IUnknown
  - AddRef
  - Release
  - QueryInterface

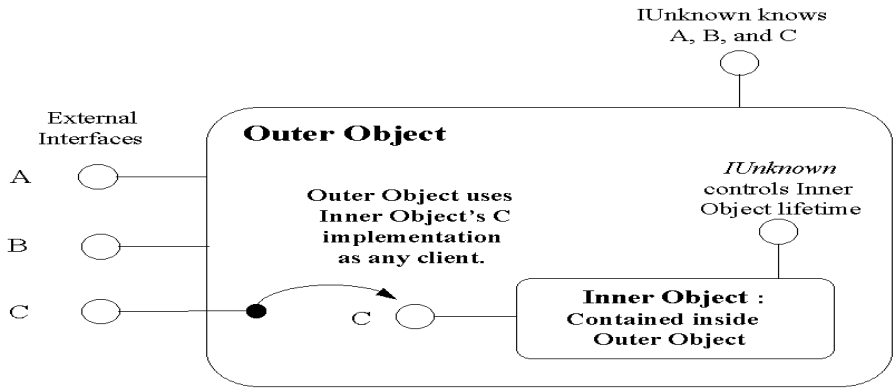
  } Functions

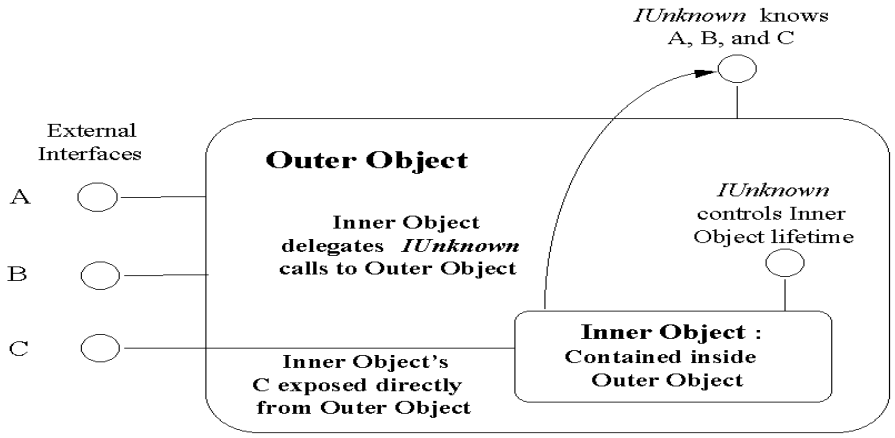
- IDispatch
  - GetIDsOfNames
  - GetTypeInfo
  - GetTypeInfoCount
  - Invoke

  } Functions

- Custom Interfaces

Interfaces:

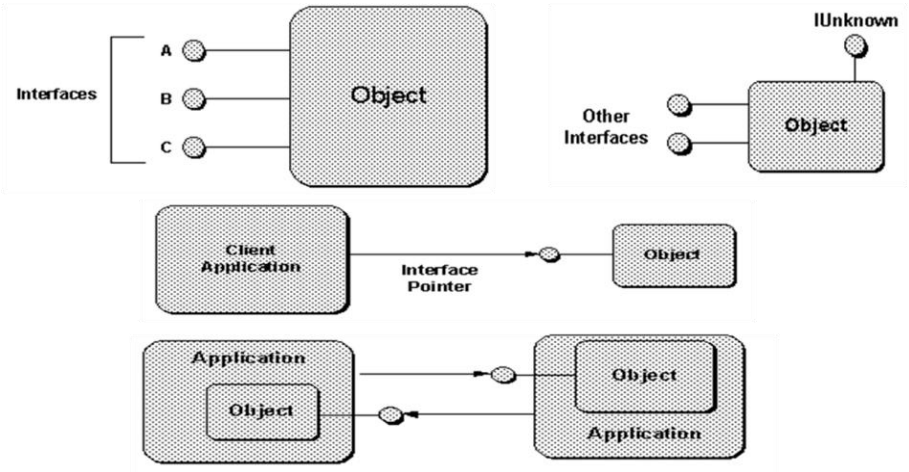




4.6 QUERRY INTERFACE

QueryInterface( )

- QueryInterface, used by clients to inquire if a component supports other specific interfaces
- QueryInterface allows clients to dynamically discover whether or not an interface is supported by a component object; if so, it returns an interface pointer from a component object
- which by specifying an IID allows a caller to retrieve references to the different interfaces the component implements. The effect of QueryInterface() is similar to dynamic_cast<> in C++ or casts in Java and C#
- QueryInterface is essentially a run-time cast – it allows you to ask a component if it implements a specific interface
  - If it does, it returns a pointer to that interface pointer in ppv
  - Think of it as a compiler/language neutral dynamic_cast operation



- COM requires that:
  - all calls to QueryInterface for a given interface must return the same pointer value
  - the set of interfaces accessible from QueryInterface must be fixed
  - if a client queries for an interface through a pointer to that interface the call must succeed
  - if a client using a pointer for one interface successfully queries for a second interface the client must be able to successfully query through the second interface pointer for the first interface
  - if a client successfully queries for a second interface and, using that interface pointer successfully queries for a third interface, then a query using the first interface pointer for the third interface must also succeed.

IUnknown Interface

- Provides three methods:
  - HRESULT QueryInterface(IID iid, void **ppv)
  - ULONG AddRef(void);
  - ULONG Release(void);
- AddRef and Release are for resource management (reference counting)

HRESULT

- This is language neutral – so no exceptions. HRESULTS are a packed bit field return value used all over COM.
- The most used return value is defined as S_OK (success, ok), the other is E_FAIL (error, failure) but there are others

- There are macros SUCCEEDED() and FAILED() that take an HRESULT and report success or failure.

Iunknown - QueryInterface Example

NOTE: Example for implementing and using QueryInterface

Listing can be divided into three sections

- First section
  - Interface IX,IY and IZ are declared
  - Interface IUknown is declared in the Win32 SDK header UNKNWN.H
- Second section
  - Implementation of the component
  - class CA implements a component supporting the IX and IY interfaces
  - function CreateInstance is defined at the end of class CA
  - client uses this function to create the component represented by CA and to return a pointer to its IUnknown
- Third section
  - main function which represents the client

CODING:

```
# include <iostream.h>
#include<objbase.h>

void trace(const char *msg )                    //Function
{
  cout<<msg<<endl;
}

// Interfaces
interfaces IX  :  IUnknown
{
 virtual void _ _stdcall  Fx( )=0;
 };
```

```
interfaces IY  :  IUnknown
{
 virtual void _ _stdcall  Fy( )=0;
 };

interfaces IZ  :  IUnknown
{
 virtual void _ _stdcall  Fz( )=0;
 };

// Forward references for GUID
extern const IID IID_IX;
extern const IID IID_IY;
```

// Component - implementation of the component

//IUnknown implementation

// IUnknown contains a member function named QueryInterface through which the client can discover whether a component supports a particular interface. If the component supports the interface, QueryInterface returns a pointer to the interface. If the component doesn't support the interface, QueryInterface returns an error code

//QueryInterface takes two parameters - first parameter identifies the interface, second parameter address where QuerInterface places the requested interface pointer

```
  class CA : pubic IX, public IY
  {
     virtual HResult _ _stdcall QueryInterface(const  IID&  iid,  void  **ppv );
     virtual ULONG _ _stdcall AddRef( ) { return 0; }
     virtual ULONG _ _stdcall Release( ) { return 0; }

// Interface IX implementation
virtual void  _ _stdcall Fx( )
{
    cout<< "Fx"<<endl;
  }

// Interface IY implementation
virtual void  _ _stdcall Fy( )
 {
   cout<< "Fy"<<endl;
    }
};
```

```
HRESULT  _ _stdcall CA :: QueryInterface(const  IID&  iid,  void** ppv)
{
    if (iid  == IID_IUnknown)
   {
           trace("QueryInterface: Return pointer to IUnknown");
          **ppv = static_cast<IX *> (this);
     }

else   if  ( iid  ==  IID_IX )
 {
  trace("QueryInterface: Return pointer to IX");
  **ppv = static_cast<IX *> (this);
  }

else   if ( iid  ==  IID_IY )
{
  trace("QueryInterface: Return pointer to IY");
  **ppv = static_cast<IY *> (this);
  }

else
{
  trace("QueryInterface: interface not supported");
  **ppv = NULL;
  Return E_NOINTERFACE;
  }
Static_cast<IUnknown*> (*ppv) -> AddRef ( )
return  S_OK;
}


// Create Instance
// function name  Create Instance which creates the component and returns a IUnknown
pointer

IUnknown *   Create Instance()
{
   IUnknown *pI = static_cast<IX *> (new CA);
   pI->AddRef();
  return pI;
}

// IID
```

```
static const IID IID_IX ={0*32bb8320, 0*b41b, 0*l1cf,
{0*a6, 0*bb, 0*0, 0*80, 0*c7, 0*b2, 0*d6, 0*82) } };

static const IID IID_IY = {0*32bb8321, 0*b41b, 0*l1cf,
{0*a6, 0*bb, 0*0, 0*80, 0*c7, 0*b2, 0*d6, 0*82) } };

static const IID IID_IZ =
{0*32bb8322, 0*b41b, 0*l1cf,
{0*a6, 0*bb, 0*0, 0*80, 0*c7, 0*b2, 0*d6, 0*82) } };


// Client

int main( )
{
 HRESULT hr;
trace (“Client :  Get an IUnknown pointer”);
IUnknown  *pIUnknown = CreateInstance( );

trace(“Client: Get interface IX”);
IX *pIX = NULL;
hr = pIUnknown -> QueryInterface(IID_IX, (void**) &pIX);

if (SUCCEEDED(hr))
{
        trace  (“Client :  succeeded getting IX”);
        pIX ->Fx( );                                    // Use Interface IX
  }

trace(“Client: Get interface IY”);
IX *pIY = NULL;
hr = pIUnknown -> QueryInterface(IID_IY, (void**) &pIY);

if (SUCCEEDED(hr))
{
        trace  (“Client :  succeeded getting IY”);
        pIY ->Fy( );                                    // Use Interface IX
  }

else        / / could not get interface

IUnknown *pIUnknownFriom IY =NULL;
(or)
IUnknown *pIUnknownFriom IY =NULL;
```

```
delete pIUknown;        // Delete the component
return 0;
}
```

4.7 Reference Counting

- AddRef and Release implement a memory management technique called reference counting
- Reference counting is a simple and fast method for enabling components to delete themselves
- COM component maintains a number called the reference count

  -When a client gets an interface from a component the reference count is incremented

  -When that client is finished using an interface , the reference count is decremented

  -When the reference count goes to 0, the component deletes itself from memory

Reference Counting Rules

- Call AddRef before returning – functions that return interfaces should always call AddRef on the pointer before returning
- Call Release when done – when finished with an interface call release on that interface
- Call AddRef after assignment - assign an interface pointer to another interface pointer call Add Ref. Increment the reference count whenever create another reference to the interface

Instantiating Objects

- interfaces (defined in IDL) for the components available in a library/on the system
- How do we actually obtain an instance of an object we want to use?
  - In COM this is termed Activation – there are three basic types, and each involves the SCM (service control manager)

Activation and the SCM

- The SCM manages the mapping between IIDs, CLSIDs, and implementations

- You can ask the SCM for a particular CLSID and it will instantiate an instance and return it's interface pointer.
  - CoGetClassObject()
  - There's an additional layer of indirection through ProgIDs – strings of the form libraryname.classname.version that map to CLSIDs
- Sometimes you want "an implementation of the following interface that meets some set of constraints"
  - enter category IDs (CATIDs)
  - You can define a set of categories, and each COM class can advertise the categories it implements.

Reference Counting rules:

Client must reference count different interface pointers even if they have nested lifetimes

1) Out parameter rule
   - function parameter that returns a value to the function's caller
   - second parameter to QueryInterface is an example of an out parameter
2) In parameter rule
   - Passes value to a function
3) In-Out parameter rule
   - Passed in and then changes the value, which passed back to the caller
4) Local variable rule
   - Local copies of interface pointers that are known to exist only for the life time of function don't require AddRef and Release pairs. This rule is a direct result of the in parameter rule.
5) Global variable rule
   - Call AddRef on the interface pointer before passing it to another function. Since the variable is global any function can end its life time by calling Release on it

NOTE: Example for Reference Counting

OBSERVE: *Coding changes for Reference Counting given in ITALIC Font style and Font size bigger*

- Listing can be divided into three sections

- First section
  - Interface IX,IY and IZ are declared
  - Interface IUknown is declared in the Win32 SDK header UNKNWN.H
- Second section
  - Implementation of the component
  - class CA implements a component supporting the IX and IY interfaces
  - *Note:* *used constructor and destructor*

    For increment and decrement AddRef and Release methods coding included
  - function CreateInstance is defined at the end of class CA
  - client uses this function to create the component represented by CA and to return a pointer to its IUnknown
- Third section
  - main function which represents the client

CODING:

```
# include <iostream.h>
#include<objbase.h>

void trace(const char *msg )                    //Function
{
  cout<<msg<<endl;
}

// Interfaces

interfaces IX  :  IUnknown
{
 virtual void _ _stdcall  Fx( )=0;
 };

interfaces IY  :  IUnknown
{
 virtual void _ _stdcall  Fy( )=0;
 };
```

```
interfaces IZ  :  IUnknown
{
 virtual void _ _stdcall  Fz( )=0;
 };

// Forward references for GUID
extern const IID IID_IX;
extern const IID IID_IY;
```

// Component - implementation of the component
//IUnknown implementation
// IUnknown contains a member function named QueryInterface through which the client can discover whether a component supports a particular interface. If the component supports the interface, QueryInterface returns a pointer to the interface. If the component doesn't support the interface, QueryInterface returns an error code
//QueryInterface takes two parameters - first parameter identifies the interface, second parameter address where QuerInterface places the requested interface pointer

```
 class CA : pubic IX, public IY
  {
     virtual HResult _ _stdcall QueryInterface(const  IID&  iid,  void  **ppv );
     virtual ULONG _ _stdcall AddRef( ) { return 0; }
     virtual ULONG _ _stdcall Release( ) { return 0; }

// Interface IX implementation
virtual void  _ _stdcall Fx( )
{
    cout<< "Fx"<<endl;
  }
// Interface IY implementation
virtual void  _ _stdcall Fy( )
 {
   cout<< "Fy"<<endl;
    }    };
public:

// Constructor

CA( ): m_cRef(0) { }
```

```
// Destructor

~CA ( ) { trace("CA:  Destroy itself");}

HRESULT  _ _stdcall CA :: QueryInterface(const  IID&  iid,  void** ppv)
{
    if (iid  == IID_IUnknown)
   {
          trace("QueryInterface: Return pointer to IUnknown");
         **ppv = static_cast<IX *> (this);
     }

else   if  ( iid  ==  IID_IX )
 {
  trace("QueryInterface: Return pointer to IX");
  **ppv = static_cast<IX *> (this);
  }

else   if ( iid  ==  IID_IY )
{
  trace("QueryInterface: Return pointer to IY");
  **ppv = static_cast<IY *> (this);
  }

else
{
  trace("QueryInterface: interface not supported");
  **ppv = NULL;
  Return E_NOINTERFACE;
  }
Static_cast<IUnknown*> (*ppv) -> AddRef ( )
return  S_OK;
}
ULONG _stdcall CA::AddRef( )
{
 cout<<"CA:  AddRef ="<< m_cRef+1<<"."<<endl;
return InterlockedIncrement(&m_Ref)==0)
}

ULONG _stdcall CA::Release( )
{
 cout<<"CA:  Realease ="<< m_cRef-1<<"."<<endl;
```

```
if(InterlockedIncrement(&m_cRef)==0)
{
 delete this;
 return 0;
}
return m_cRef
}

// Create Instance
// function name  Create Instance which creates the component and returns a IUnknown
pointer

IUnknown *   Create Instance()
{
   IUnknown *pI = static_cast<IX *> (new CA);
   pI->AddRef();
  return pI;
}


// IID
static const IID IID_IX =
{0*32bb8320, 0*b41b, 0*l1cf,
{0*a6, 0*bb, 0*0, 0*80, 0*c7, 0*b2, 0*d6, 0*82) } };

static const IID IID_IY =
{0*32bb8321, 0*b41b, 0*l1cf,
{0*a6, 0*bb, 0*0, 0*80, 0*c7, 0*b2, 0*d6, 0*82) } };

static const IID IID_IZ =
{0*32bb8322, 0*b41b, 0*l1cf,
{0*a6, 0*bb, 0*0, 0*80, 0*c7, 0*b2, 0*d6, 0*82) } };


// Client

int main( )
{
 HRESULT hr;
trace (“Client :  Get an IUnknown pointer”);
IUnknown  *pIUnknown = CreateInstance( );

trace(“Client: Get interface IX”);
IX *pIX = NULL;
```

```
hr = pIUnknown -> QueryInterface(IID_IX, (void**) &pIX);

if (SUCCEEDED(hr))
{
        trace  (“Client :  succeeded getting IX”);
        pIX ->Fx( );
         pIX-> Release( );                          // Use Interface IX

  }
trace(“Client: Get interface IY”);
IX *pIY = NULL;
hr = pIUnknown -> QueryInterface(IID_IY, (void**) &pIY);
if (SUCCEEDED(hr))
{
        trace  (“Client :  succeeded getting IY”);
        pIY ->Fy( );                                  // Use Interface IX
  }
else        / / could not get interface
IUnknown *pIUnknownFriom IY =NULL;
(or)
IUnknown *pIUnknownFriom IY =NULL;

Trace (“Client : Realse IUnknown interface”)

pIUnknown ->Release( );                          // Delete the component
      return 0;
}
```

COM Class Factories

- A COM class object is a component that creates new instances of other objects.
- Class objects implement the IClassFactory interface and are called class factories. IClassFactory interface has two methods:
  - CreateInstance accepts an interface identity number and returns a pointer, if successful, to a new component object.
  - LockServer turns on, or off, locking of the factory’s server in memory.

COM instantiates factories using a global or static member function provided by the factory code:

```
DllGetClassObject(REFCLSID clsid, REFIID riid, void **ppv)
```

Standard COM Interfaces

- IClassFactory is used by COM to create instances of the component

IClassFactory
CreateInstance(IUnknown *pUnknownOuter, REFIID riid, void **ppv) LockServer(BOOL bLock)

- IUnknown is used by Clients to get interface pointers

IUknown
QueryInterface(REFIID riid, void **ppv) Addref( ) Release( )

There are many other standard COM interfaces

CLASS FACTORY

- Class factory is the component that creates a component
- Class factory for a component generally implements IClassFactory
- Some components have special requirements when they created , these components will implement other interfaces instead of or in addtion to IClassFactory
- COM library contains the function named CoCreateInstance for creating components
- Component created with CoCreateInstance are created through IClassFactory interface
- CoCreateInstance creates a component called class factory, which then creates a desired component
- When CoCreateInstance isn't enough so then use CoGetClassObject (to get direct control over the class factory and the interface used to create the component)

➢ CoGetClassObject

- want to create an object using a creating interface other than IClassFactory
- if want to create a bunch of components all at one time

Parameters

- Both functions take three parameters
- CLSID of the desired component as the first parameter

- Last two parameter

  CoCreateInstance - returns a pointer to the component itself

  CoGetClassObject – returns the requested pointer to the class factory

NOTE:

1) Two functions differ only in a single parameter,

   CoCreateInstance takes an Iunknown pointer

   CoGetClassObject takes a COSERVERINFO pointer (COSERVERINFO pointer is used by

   DCOM to control accessing remote components)

2) Both function also take the execution context, *dwClsContext*

CoCreateInstance Declaration

```
HRESULT  _stdcall CoCreateInstance
{
  const CLSID& clsid,
  IUnknown  *pIUnknownOuter          //Outer component
  DWORD dwClsContext,                 // Server context
  const IID&  iid,
  void **ppv
  };
```

NOTE:
- four in parameters and single out parameter
- *dwClsContext* restricts the execution context of the components to which the client can connect

CoGetClassObject Declaration

```
HRESULT _stdcall CoGetClassObject
{
 const CLSID& clsid,
  DWORD dwClsContext,                  // Server context
  COSERVERINFO *pServerInfo          // Reserved for DCOM
  const IID&  iid,
  void **ppv
  };
```

4.8 COM Class Factories

- A COM class object is a component that creates new instances of other objects.
- Class objects implement the IClassFactory interface and are called class factories. IClassFactory interface has two methods:
  - CreateInstance accepts an interface identity number and returns a pointer, if successful, to a new component object.
  - LockServer turns on, or off, locking of the factory's server in memory.
- LockServer

  - Provides the client a way to keep the server in memory until it finish (client simply calls LockServer(TRUE) to lock the server, LockServer(FALSE) to release the server

- COM instantiates factories using a global or static member function provided by the factory code:

  DllGetClassObject(REFCLSID clsid, REFIID riid, void **ppv)

IClassFactory has two member function

- CreateInstance
- LockServer

- IClassFactory::CreateInstance

  - has three parameters

First parameter is pointer to Iunknown interface

Other two parameters for client request an interface

Eg.,

```
interface IClassFactory:Iunknown
{
HRESULT   _stdcall CreateInstance(Iunknown *pUnknownOuter, const IID& iid, void
                                                                          **ppv)
HRESULT  _stdcall LockServer(BOOL block);
};
```

Implementing the Class Factory

- Using DllGetClassObject
- Same three parameters as same CoGetClassObject
- Passing CLSID to DllGetClassObject
- This parameter allow single DLL to support any number of components

Registering the component

- Functions in windows registry

  DllRegisterServer register component

  DllUnregisterServer unregister component

Unloading the DLL

- COM Library implements a function named CoFreeUnused Libraries
- CoFreeUnused Libraries can unload DLL
- Client periodically call CoFreeUnused Libraries during idle time
- DllCanUnloadNow

  - CoFreeUnused Libraries asks the DLL whether unloaded by calling DllCanUnloadNow

  - If DLL is not serving any objects, Use DllCanUnloadNow

- To determine whether it is still serving components, DLL keeps a count of them (checks count for 0)

- LockServer

- Provides the client a way to keep the server in memory until it finish (client simply calls LockServer(TRUE) to lock the server, LockServer(FALSE) to release the server

Class Factory example

Write for Class implementing IClassFactory

```
Class CFactory: public IClassFactory
{

// use code as similar to reference counting

// Interface IClassFactory
virtual HRESULT _stdcall CreateInstance(IUnknown  * pUnknownOuter, cost IID&, iid,
void **ppv);

virtual HRESULT _stdcall  LockServer(BOOL block);

 // class factory IUnknown implementation
  ULONG _stdcall CFactory::AddRef()
  {
       return InterlockedIncrement(&m_cRef);
     }
 ULONG _stdcall CFactory::Release()
 {
   if ( InterlockedDecrement(&m_cRef)==0;
        {
          delete this;
          return 0;
         }
    return m_cRef;
      }


      // IClassFactory implementation

      // Lock Server
```

// Exported Function

// Server registration

// server Unregistration

// DLL module information

Example 2:

Note:

- Global variab;e section for modules, version etc.,
- Component coding implementing using constructor and desctructor
- IClassFactory code for AddRef( ) and Release ( )
- IClassFactory implementation

```
# include <iostream.h>
#include<objbase.h>
#include “Iface.h”     // Interface declaration
#include “Registry.h”   // Registry helper functions

void trace(const char *msg)
{
  cout<<msg<<endl;
}

// Global variables

static HMODULE g_hModule =NULL;        // DLL module handle

static long g_cComponents =0;

static long g_cServerLocks =0;

// Interfaces

interfaces IX:IUnknown

{

 virtual void _stdcall Fx( )=0;
```

```
 }
interfaces IY:IUnknown
{
 virtual void _stdcall Fx( )=0;
 }

// Forward references for GUID
extern const IID IID_IX;
extern const IID IID_IY;

/ / Component
class CA : pubic IX, public IY
{
//IUnknown implementation
virtual HResult _stdcall QueryInterface(const IID& iid, void  **ppv
virtual ULONG _stdcall AddRef() { return 0}
virtual ULONG _stdcall Release() { return 0}

// Interface IX implementation
virtual void  _stdcall Fx() { cout<< “Fx”<<endl;
// Interface IY implementation
virtual void  _stdcall Fy() { cout<< “Fy”<<endl;

HRESULT  _stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
 if (iid  == IID_IUnknown)
{
```

```
  trace("QueryInterface: Return pointer to IUnknown");
  **ppv = static_cast<IX *> (this);
  }
else  if (iid  == IID_IX)
{
  trace("QueryInterface: Return pointer to IX");
  **ppv = static_cast<IX *> (this);
  }

else  if (iid  == IID_IY)
{
  trace("QueryInterface: Return pointer to IX");
  **ppv = static_cast<IY *> (this);
  }
else
  trace("QueryInterface: interface not supported");
  **ppv = NULL;
return  S_OK;

}
        // IClassFactory implementation
        // Lock Server
        // Exported Function
        // Server registration
        // server Unregistration
        // DLL module information
```

Refer to page no. 146 to 149 for more details on Classfactory in “INSIDE COM” by Dale Rogerson

4.9 DYNAMIC LINK LIBRARIES

DLL and EXE

- DLL stands for dynamic-link library
  - Microsoft’s implementation of a shared library
  - This means that many different programs can use this library to do their tasks making it easier on the programmers so that they do not have to keep reinventing the wheel each time they write software. In simple terms a .DLL file will contain logic that other programs will use

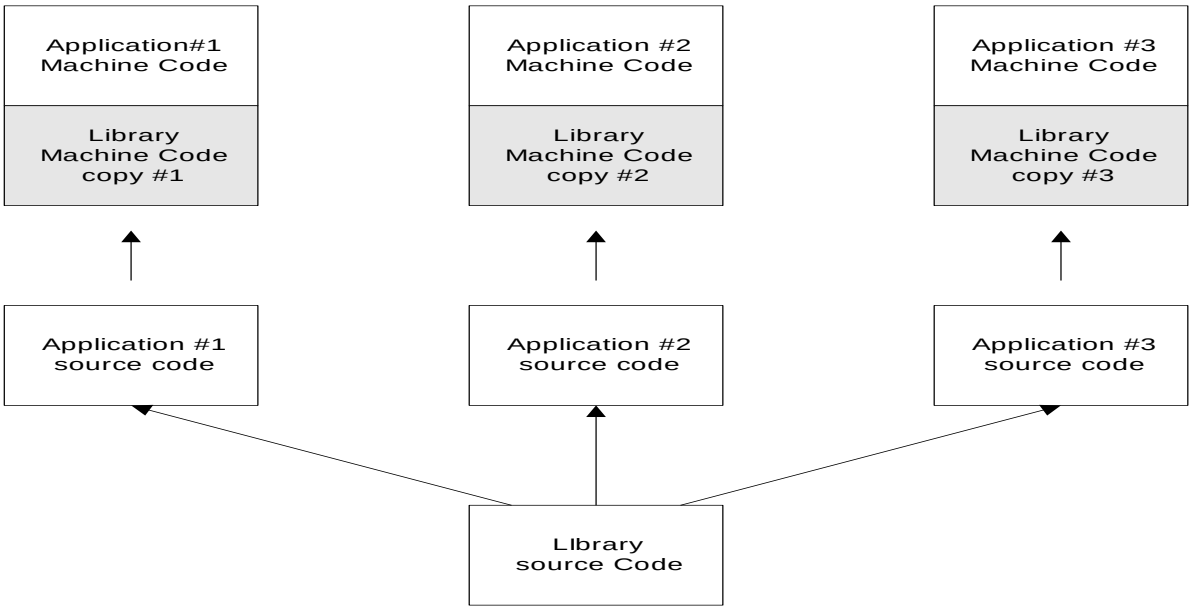
- EXE stands for executable
  - Denotes that a program is executable. This just means that if you double click on the file a program will run, normally with some kind of interface for a user to interact with.
  - The file formats for EXE and DLL actually the same
- In windows an executing program is known as process
- **About exe**
  - every applications exes runs in a separate process
  - address in one processes is different from an address in another process
  - pointer’s can be passed from one application to another because they reside in different address spaces
  - For this reason out processes
  - pointers in two processes can seem to contain the same address but actually refer to different physical memory locations

- **About dll**
  - dynamic link library resides in same process as the application to which it is linked.
  - For this reason, called in-process servers

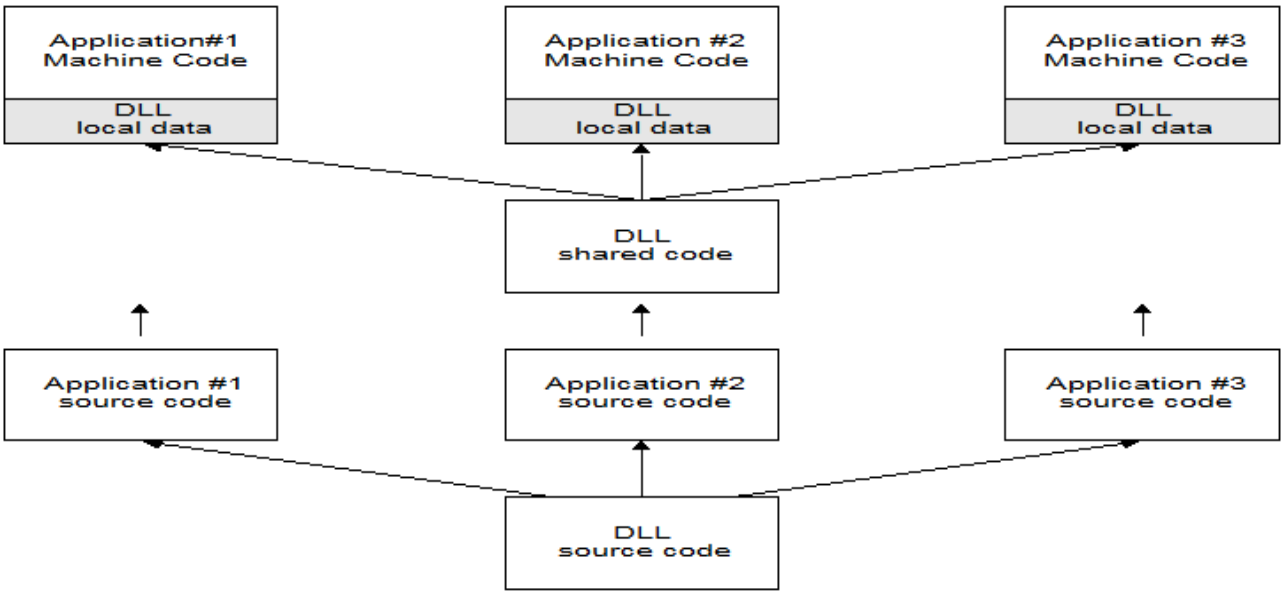
Code Reuse by Using DLLs

- Use dynamic link libraries (DLLs)
  - DLLs are loaded at run time from a single file into any running program that needs them, saving disk space for one copy of the object code for each executable that uses the library
  - DLLs used by several concurrently running executables have only one copy of their code in memory, although each executable maintains local storage for the DLL code. This saves RAM space that would otherwise be required for each running program using the DLL

Duplication of Library Code with Static Linking



Sharing of DLL Code



NOTE: Example for creating dll and import (adding two numbers)

Creating DLL

C and C++

```
#include <windows.h>

 // DLL entry function (called on load, unload, ...)

BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved)
{
   return TRUE;
}


// Exported function - adds two numbers
extern "C" __declspec(dllexport) double AddNumbers(double a, double b)
{
   return a + b;
}
```

Delphi

```
library Example;

 // function that adds two numbers
```

```
function AddNumbers(a, b : Double): Double;
begin
  Result := a + b;
end;

// export this function
exports AddNumbers;

// DLL initialization code: no special handling needed
begin
end.
```

Using DLL imports

C and C++

```
#include <windows.h>
#include <stdio.h>

// Import function that adds two numbers
extern "C" __declspec(dllimport) double AddNumbers(double a, double b);

int main(int argc, char *argv[])
{
    double result = AddNumbers(1, 2);
    printf("The result was: %f\n", result);
    return 0;
}
```

Using explicit run-time linking

Microsoft Visual Basic

```
Option Explicit

    Declare Function AddNumbers Lib "Example.dll" _
    (ByVal a As Double, ByVal b As Double) As Double
```

```
Sub Main()
    Dim Result As Double
    Result = AddNumbers(1, 2)
    Debug.Print "The result was: " & Result
End Sub
```

Compare COM and DCOM

- DCOM transparently expands the concepts and services of COM.
- DCOM builds on the client-side proxy objects and the server side stub objects already present in COM.
- In COM, the request and responses are delivered via the *Lightweight Remote Procedure Calls (LRPC)*.
  - DCOM supports clients and servers residing on separate nodes (remote server).
- In DCOM, the mechanism for request and response remains the same, except DCOM replaces LRPC with *Object-Oriented RPC (ORPC)* that uses network protocol, developed upon the base of DCE remote procedure calls from OSF. Neither the client nor the component is aware that the wire that connects them has just become a little longer.

- In both COM and DCOM, remote method calls are synchronous.

Comparison of DCOM, CORBA, RMI

Java RMI comes from JavaSoft

CORBA is a specification resulting from OMG

DCOM comes from Microsoft

Basic principles

	CORBA	**RMI**	**COM / DCOM**
Language	Many	Java	C++, Java, VB, …

Remote reference	Object reference	object ID	Handles to remote Interface (Interface Pointer)
IDL Interface	OMG IDL	JAVA	MIDL
Proxy	IDL stub IDL skeleton	stub, skeleton	proxy, stub
Marshalling	IIOP	Serialization	Network data representation(RPC)

Differences between DCOM and CORBA

Differences	DCOM	CORBA
Focus	Desktop first; enterprise second	Enterprise first; desktop second
Platforms	Windows NT; future support for Windows (all), Macintosh, UNIX, MVA	MVS, UNIX, Windows (all), Macintosh
Availability	Single vendor; availability from other vendors expected	Multi-vendor
Service differences	ActiveX-interactive content standard	Significant number of additional services, including query, trader, transactions, as well as facilities in the areas of information management and system management. Lastly, services in areas such as finance, distributed simulation, and computer integrated manufacturing

Differentiate DCOM and CORBA

	DCOM	CORBA
Top layer: Basic programming architecture		
Common base class	`IUnknown`	`CORBA::Object`
Object class identifier	`CLSID`	interface name
Interface identifier	`IID`	interface name
Client-side object activation	`CoCreateInstance()`	a method call/`bind()` [footnote 5]
Object handle	interface pointer	object reference
Middle layer: Remoting architecture		
Name to implementation mapping	Registry	Implementation Repository
Type information for methods	Type library	Interface Repository
Locate implementation	SCM	ORB
Activate implementation	SCM	OA
Client-side stub	proxy	stub/proxy
Server-side stub	stub	skeleton
Bottom layer: Wire protocol architecture		
Server endpoint resolver	OXID resolver	ORB
Server endpoint	object exporter	OA
Object reference	OBJREF	IOR (or object reference)
Object reference generation	object exporter	OA
Marshaling data format	NDR	CDR
Interface instance identifier	IPID	object_key

UNIT V INTRODUCTION AND PROBLEM SOLVING 10 hrs. Overview – Architecture – Key technologies – UDDI – WSDL – SOAP.

Overview

1.1 Web Services architecture

We service is a software component which can be accessed via standard protocol. The protocol such as SOAP (Simple Object Access Protocol) for communication, XML for messaging, WSDL (Web Service Description Language) provides description of web service and UDDI (Universal Description Discovery and Integration) for service registration and discovery.

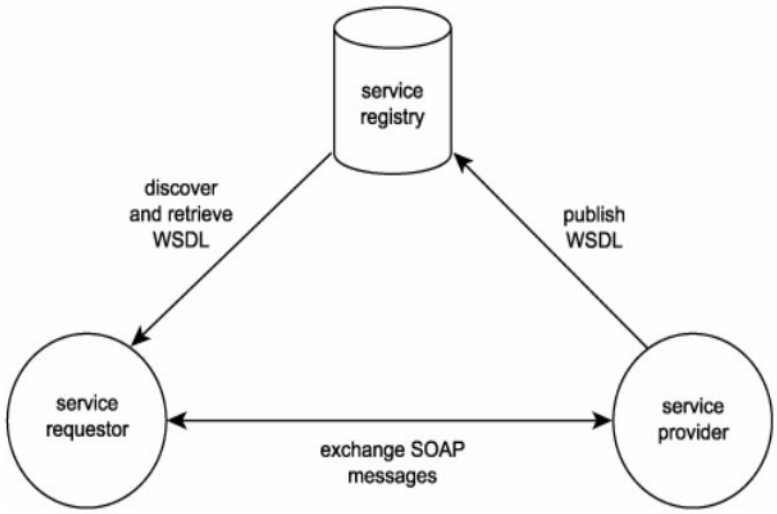


Figure 5.1 : Web service architecture

Key Technologies : Web service protocol stack

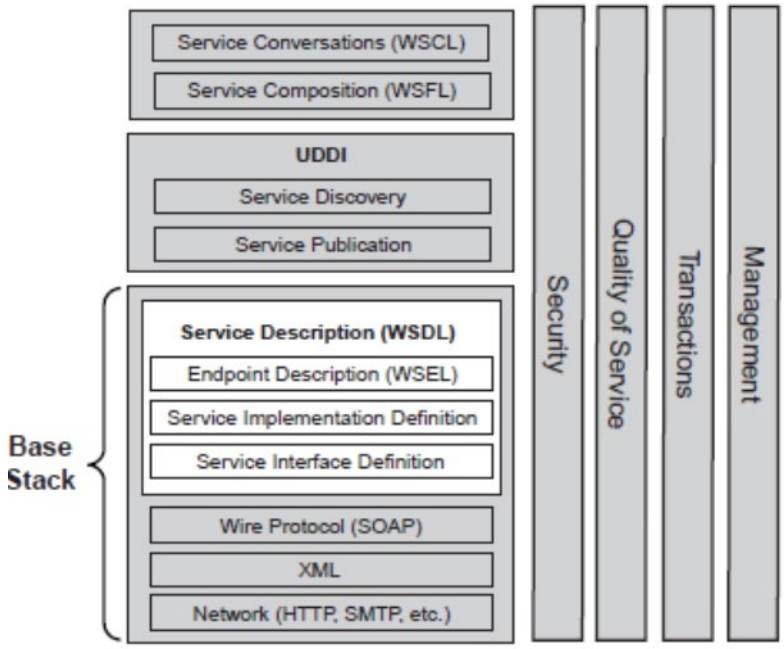


Figure 5.2 : Web service protocol stack

- Each layer on left builds upon the capabilities of the layer beneath it.
- Vertical column represents the architect must address at every level of stack.
- Base stack includes technologies necessary to create and invoke web services.
- **Network layer allows web services to be available to service requesters.**
- SOAP is XML based messaging protocol that forms the basis for all interactions with web services.
- With running on top of http SOAP messages are simple POST operations with SOAP XML envelop as the payload.
- SOAP messages support the publish , find and bind operations that forms the basis of SOA.
- WSDL – Web Service Description language
- WSEL → Web Service End point Description Language

Service Interface definition → contains binding ,Port Type, message and Type elements which forms portions of service description that is reusable from one implementation to another.

Service Implementation Definition

- Contains the elements that are specific to each implementation.

- The Service and port elements
- A third party (standard body) might specify the service interface definition for a particular type of web service.
- End point description → which introduces semantics to the service description that apply to a particular implementation.
- Contain security , QoS and management attributes that help to define the policies for each of these vertical columns.
- Development team uses UDDI to publish the services to a registry or another repository of information about available web service.
- Once the architect has dealt with issues of service publication and discovery , he can move the complex issues of interaction of multiple web services.
- WSFL and WSCL are still in development

Security ,QoS ,Transaction and service management

- Each apply to every layer in the stack.
- The architect must know --| (each vertical column with horizontal layer)
- Security of individual messages encrypted by encrypted payloads and signatures which are encompasses to SOAP headers
- Securing web services in publication and discovery across internet is another issue.
- QoS→ network QoS involves network uptime , packet delivery, valid http messages
- Transaction is one of the issue
- Rollback conversation

Management of web services

- Handled by management application.

This application must be able to do

- Availability and health of web services infrastructure including network and physical systems that support the execution of web services.
- Determine the availability and health of service registries.
- Determine the availability and health of external web services once discovered invoked them. These services are external and may not provide management interface.

1.2 UDDI (Universal Description, Discovery and Integration)

- UDDI is a directory service where business can register and search for web services.
- UDDI stands for Universal Description Discovery and Integration.
- UDDI is a directory for storing information about webservice.
- UDDI was originally created by Microsoft, IBM and Ariba, represents a technical specification for publishing and finding business and webservices.
- UDDI consists of two parts. First, UDDI is a technical specification for building a distributed directory of business and web service.
- Data is stored within a specific XML format. The UDDI specification includes API details for searching existing data and publishing new data. Second, the UDDI business registry is a fully operational implementation of UDDI specification. The data captured within UDDI divided into three main categories.

WHITE PAGES:

White pages provide very limited information such as name, telephone and contact address.

YELLOW PAGES:

Yellow pages provide a categorization based on business and service type. For example, the data may include industry, product

GREEN PAGE:

It includes technical information about the web service.

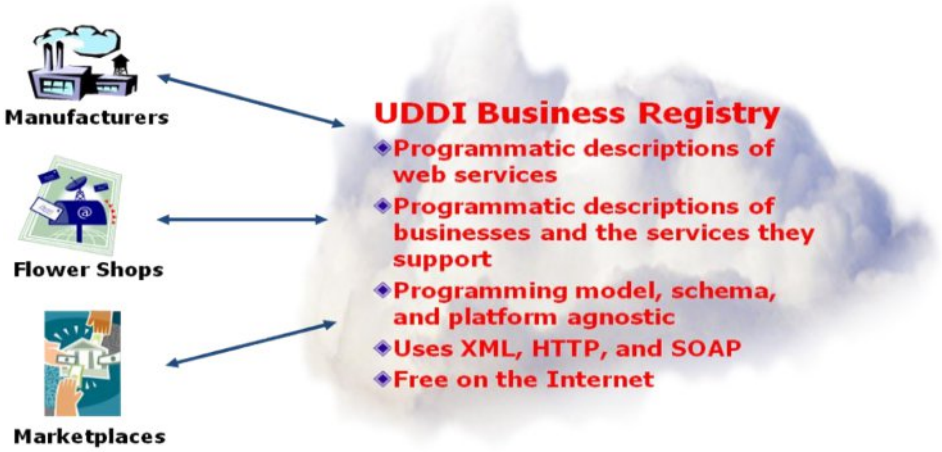
White and yellow pages directories were, earlier, available in the printed format.

In the present one, these directories are available online and are accessible globally.

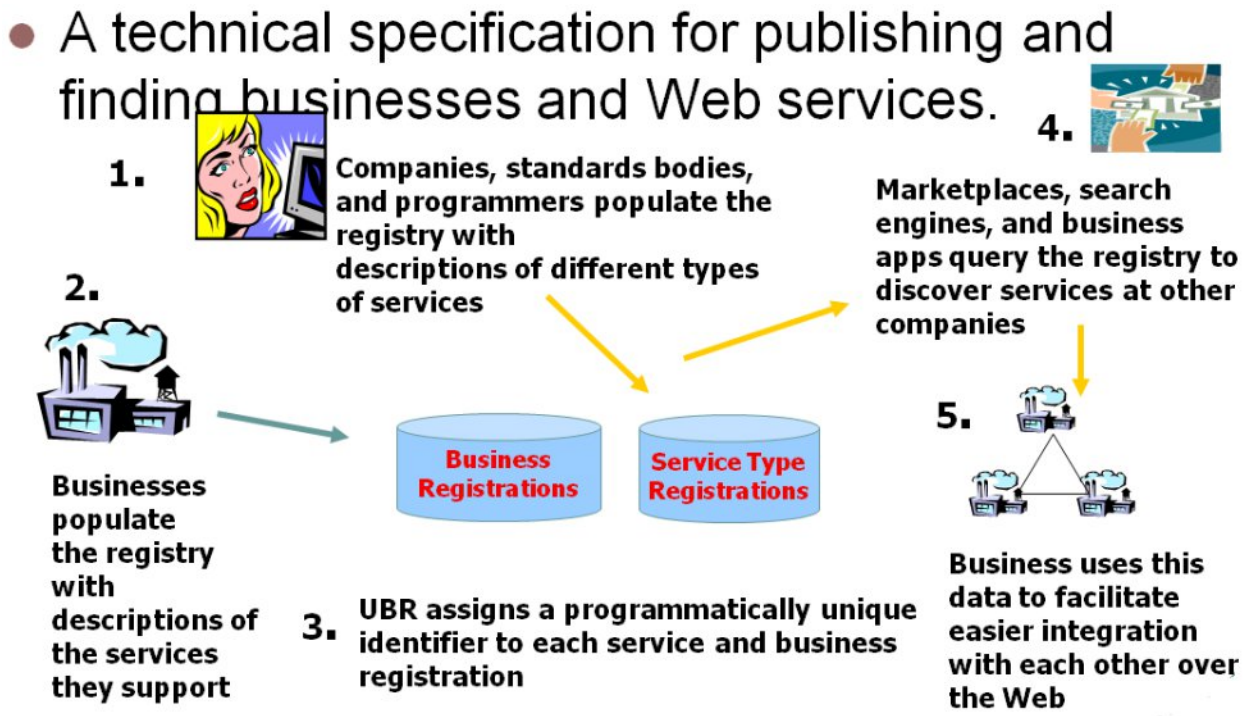
UDDI Registry

- Functional purpose: representation of data and metadata about Web services
- Either for use on a public network or within on organizational internal infrastructure
- Offers standards-based mechanism to classify, catalog and manage Web services so that they can be discovered and consumed by other applications
- Implements generalized strategy of indirection amongst services based applications
- Offers benefits to IT managers both at design and run-time, including code reuse and improving infrastructure management

UDDI Implementation



How UDDI Works



UDDI Features

- UDDI is similar to the concepts of DNS and yellow pages
- In short, UDDI provides an approach to:
    - Locate a service

- Invoke a service
- Manage metadata about a service

- UDDI specifies:
  - Protocols for accessing a *registry* for Web services
  - Methods for controlling access to the *registry*
  - Mechanism for distributing or delegating records to other *registries*

UDDI Technical Overview

1.UDDI data model - An XML Schema for describing businesses and web services.

2.UDDI API - A SOAP-based API for searching and publishing UDDI data

UDDI Data Model

UDDI describes four core types of information:

1) Business Entity

2) BusinessService

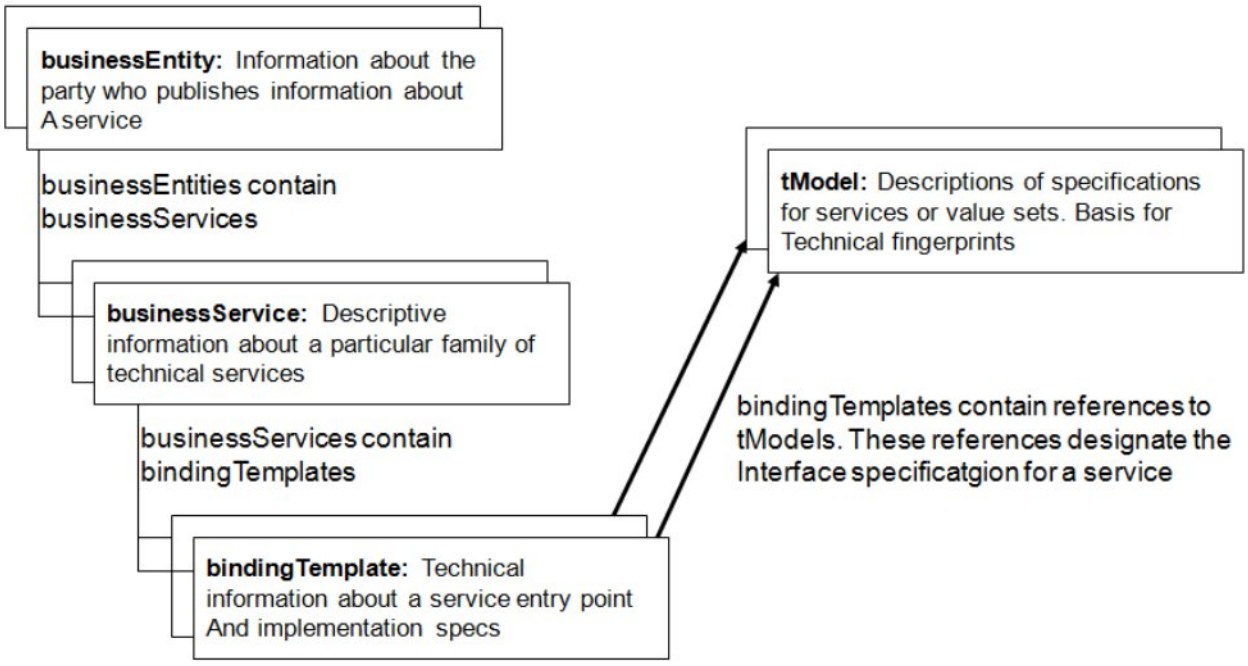
3) BindingTemplate

4) tmodel

UDDI Data Model

- UDDI describes four core types of information:
  - businessEntity
    - A business or organization providing services.
    - White page.
  - businessService
    - Services provided by an organization.
    - Support classification using various taxonomy systems.
    - Yellow page.
  - bindingTemplate
    - Technical information necessary to access a service.
    - Green page.

- tModel (Technical Model)
    - Descriptions and pointers to a reusable concept, external technical specifications or taxonomies.
    - E.g., Web service type, a protocol used by Web services, a category system.



BUSINESS ENTITY:

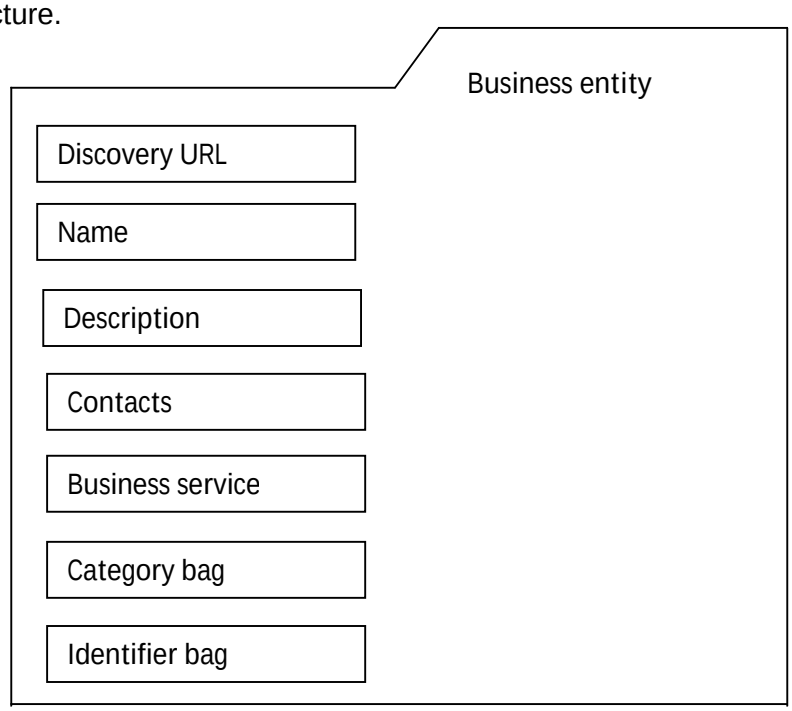
Business service the entity hosts

- category bag - contains names and values of entity's properties
- Identifier Bag - contains names and value of general information about the entity
- Contacts – list of people to contact
- DiscoveryURL – location of document, that contain more information

Business entity data structure represents the top-level UDDI element. This element holds descriptive information about business. The business entity data structure contains information such as contact information, categorization, identifiers, descriptions and business relations. The business entity data structure defines an important attribute called business key, which is a unique key.

This key is provided by organization itself at the time of registration or it could be generated by the registry while publishing the service. Information retrieval from the UDDI

registry will contains the key information. The graphical representation of the business entity data structure.



The <identifier bag> data structure contains a list of identifiers; the <category bag> data structure contains a list of business categories that describe a specific business aspect.

<businessentity businesskey="….." operator="……" authorizedname="…….">
    <name>……..</name>
    <description>…… </description>
    <contact>…… </contact>
    <businessservice>……..</businessservice>
    <categorybag>
        <keyedReference tmodelkey="…." Keyname="……." keyvalue="……"/>
    </categorybag>
    <identifierbag>
        <keyedReference tmodelkey="…." Keyname="……." keyvalue="……"/>
    </identifierbag>
</businessentity>

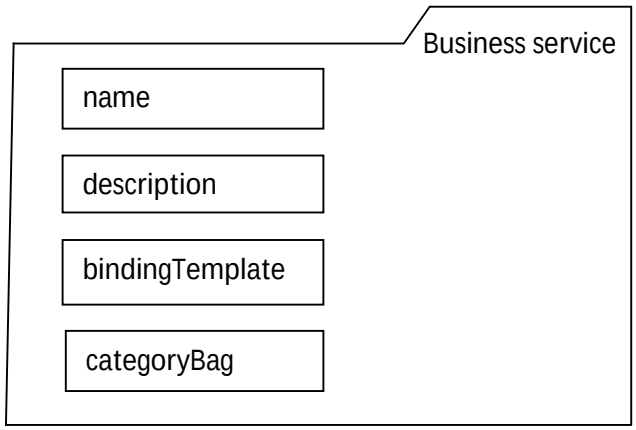
Business Service:

It contains

- Service Key – a GUID that uniquely identify the service

- Name – human readable name for the service
- Description - human readable description for the service
- Binding Template – set of properties that define the taxonomy of the service
- CategoryBag – a name- value pair that helps to define categories to which the service belongs

Business service data structure provides information scout a single web service or group related with web services. The business service data structure contains descriptive information such as the name of the service, description, classification. The graphical representation of the business service is,



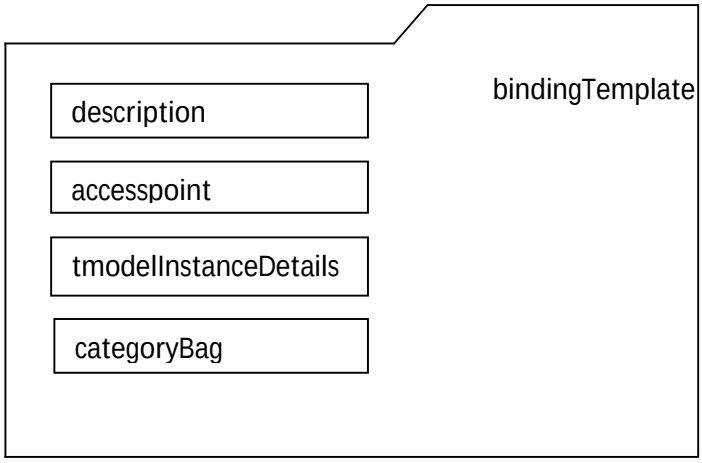
The data structure defines two important attributes, business key and service key.

<businessService
servicekey=" "
businessKey=" ">
<name> </name>
<description> </description>
<bindingTemplates> </bindingTemplates>
<categoryBag> </categoryBag>
</businessService>

Binding Template:

- Binding template data structure represents individual web service.
- The data structure stores a list of technical information needed by the application to find and interact with web service being described.
- The data structure defines two important data structure accesspoint and tmodelInstanceDetails

- The accespoint is string that is used to represent a suitable network address for invocation of the web service being described. This could be a URL, email address or any other textual information.
- The accesspoint supports an optional attribute called UseType.
- UDDI provides four pre-defined UseType attributes value, endpoint, bindingTemplate, hostingRedirector and wsdlDeployment.
- Among all these, wsdlDeployment is important for the understanding of web services.
- This includes that the accesspoint data structure points to WSDL document that contain necessary information for binding and service invocation.
- The tmodelInstanceDetails data structure is a mandatory structure. This data structure hosts another important data structure called tmodelInstanceInfo.
- This tmodelInstanceInfo element defines on attribute called tmodelKey.
- This is a mandatory attribute that represents a specification with which the web service that is represented by the corresponding bindingTemplate compiles.

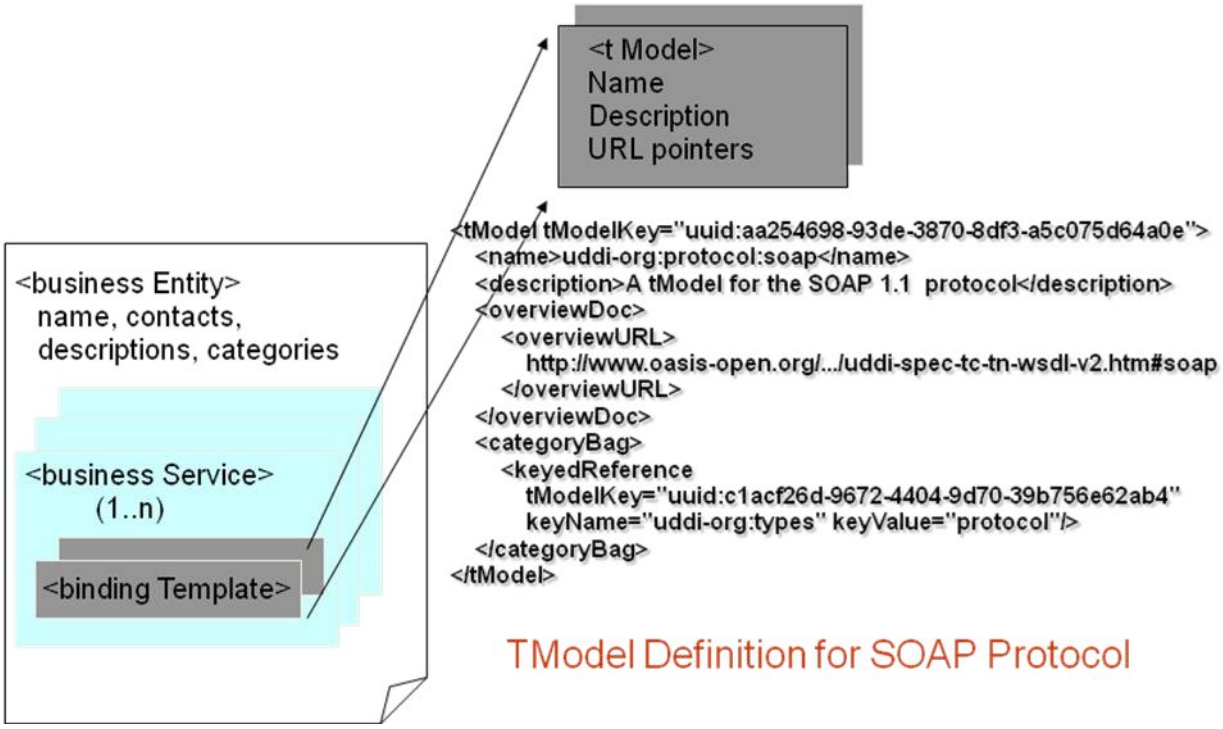



```
<bindingTemplates>
        <bindingTemplate
                serviceKey="        "
                bindingKey="        ">
       <description>          </description>
       <accessPoint      URLType=" "  >          </accessPoint>
       <tmodelInstanceDetails>
                <tmodelInstanceInfo      tmodelKey="          "/>
       </ tmodelInstanceDetails >
</bindingTemplate>
</bindingTemplates>
```

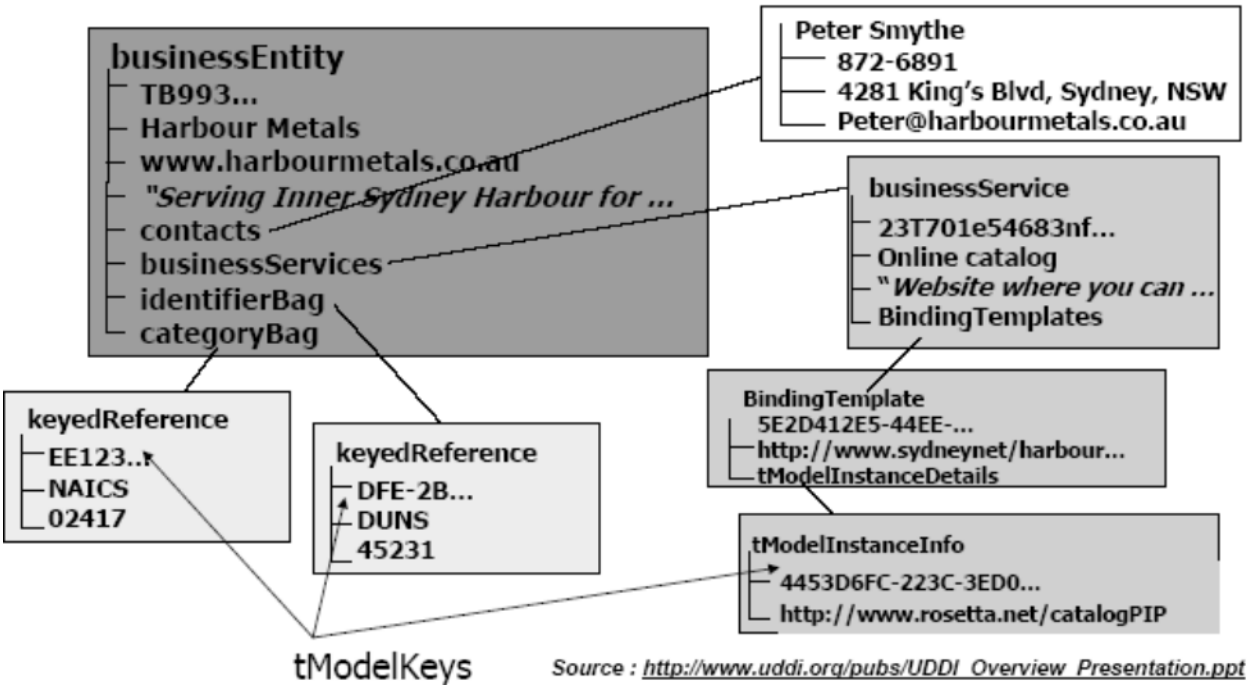
tModel

- tModel documents are a core data structure in the UDDI specification and represent the most detailed information that a UDDI registry can provide about any specification
- There are several places within a businessEntity that can refer to tModels
  - Defining the **technical fingerprint**
    - One common use for tModel entities is to represent technical specifications
    - e.g. a tModel can be used to represent a specification that defines wire protocols
  - Defining **value sets**
    - specify organizational identity and various categories
    - represents the system of values used to identify or categorize UDDI entities
  - Defining a **find qualifier**
    - Find qualifiers are values that modify how the find_xx APIs work.

Example of tModel



Example of a Registration



UDDI API:

Inquiry API:

The inquiry queries are as follows:

- find_binding: locates specific binding within a registered businessService and returns a bindingDetail message that contains a bindingTemplate elements structre. If there are no matches, the returned bindingDetail will be empty. If an error occurs, a dispositionReport structure will be returned in a SOAP Fault element.
- find_business: locates information about one or more business and returns a businessList message. Searches can be performed on name elements (or partial name elements), identifierBag elements, categoryBag elements, tModelBag elements, or discoveryURL elements. tModelBag elements are collection of tModel elements that allow searches for compatible bindings. If there are no matches, an empty businessList is returned. Errors are handled as they are with the find_binding query.
- find_service: locates specific services within a registered businessEntity and returns a serviceList message. Searches can be performed on name elements (or partial name elements), identifierBag elements, categoryBag elements, or tModelBag elements. If

there are no matches, an empty businessService structure is returned. Errors are handled as they are with the find_binding query.

- find_tModel: locates one or more tModel information structures and returns a tModelList structure, which is a list of abbreviated information about tModel elements that match the search criteria. Search parameters, no match conditions, and error conditions are handled the same as the preceding queries.

Publication API:

The publication queries are as follows

- delete_binding: removes an existing bindingTemplate from the bindingTemplates collection that is part of a specified businessService structure. The bindingTemplate is identified by its bindingKey. If successful, a dispositionReport with a single success indicator is returns.
- delete_business: deletes a business by deleting registered businessEntity structures from the registry. The businessEntity is identified by its businessKey.
- delete_service: deletes an existing businessService structure from the businessServices collection, which is part of a specified businessEntity. The businessService is identified by its serviceKey.
- delete_tModel: used to delete registered information about one or more tModel structures. If there are any references to a tModel when this call is made, the tModel will be marked as deleted, or hidden, instead of being physically removed. Hidden tModel elements can still be accessed by the owner but are not returned is search results. However, the details in a hidden tModel are still accessible, so this should be nulled out with the save_tModel call, if the owner wishes the details to be deleted.

1.3 WSDL

Describe a web service, the Web Service Description Language (WSDL) are used. WSDL is an XML based language for describing web services.

Requirements for Describing Web Services

- To send a message correctly you need to know the following
- The IP address and other end point information.
- If using http SoapAction

- The allowable set of messages (called operations) for the request message of each operations.
- The response schema to expect if there is a response.
- Any possible headers that may be expected in the request or response and the schema for those.
- Whether or not each operation is defined according to the rule of SOAP.

Describing Web Services

- WSDL describes four critical pieces of data
- Interface information describing all publically available functions.
- Data type information for all message requests and message responses.
- Binding information about the transport protocol to be used.
- Address information for locating the specified service.
- WSDL represents a contract between service requester and service provider
- WSDL is a platform independence, language independent to describe SOAP services.
- Using WSDL client can locate a web service and invoke any of its publically available functions.
- WSDL provides a common language for describing services and platform for automatically integrate those services.
- WSDL is an XML based grammar that describes how external clients interact with web method at a given URL
- WSDL document viewed as contract between the web service client and web service itself.

Basic format of a WSDL document

A valid WSDL document is opened and closed using the root <definition> element.

<definition> element will specify xml namespaces that define xml schema, SOAP elements, target namespace.

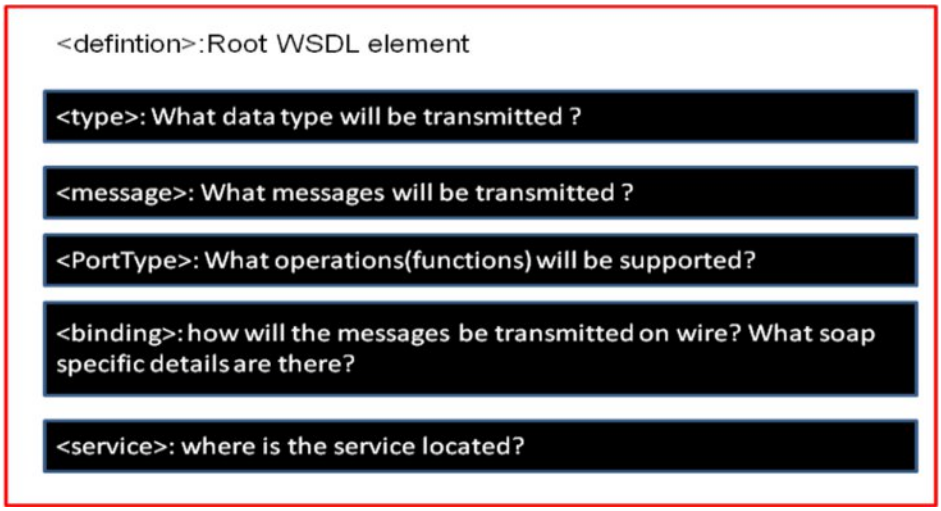
<definition>

<type>

<!– List of types exposed from WS >

```
        </type>
        <message>
            <!– format of the message>
        </message>
        <portType>
            <!- port information>
        </portType>
        <binding>
            <!– binding information>
        </binding>
        <service>
            <!– Information about the web service itself -- >
        </service>
    </definition>
```

Basic format of a WSDL document



Anatomy of WSDL-WSDL specification

- WSDL is an xml grammar for describing web services. the specification divided into six major elements.

<definition>

The defintion element must be the root element of all WSDL document.

- It defines name of the web service, declares multiple namespaces used throughout the remainder of the document, and all service element describe here.

<definitions name="HelloService"

targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"

xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"

xmlns:xsd="http://www.w3.org/2001/XMLSchema">

Types element

- Describes all the data types used between the client and server
- W3C xml schema specification as it default choice, type system not tied to any specific system.
- If the service uses only xml schema built-in simple types such as strings and integers, the type element is not required.Using

Message element

- Describes name of the message contain zero or more message part elements, which can refer to message parameter or message return values.

```
<message name="SayHelloRequest">
 <part name="firstName" type="xsd:string"/>
</message>
<message name="SayHelloResponse">
 <part name="greeting" type="xsd:string"/>
</message>
```

Part → specifies functional parameters (here firstName)

for response the part specifies function return values.

The *part* element's type attribute specifies an XML Schema data type.

Port Type

- The portType element defines a single operation
- The portType element combines multiple message element to form a complete one-way or round-trip operation.
- A portType can combine one request and one response message into a single request/response operation.
- portType can define multiple operation.

```
<portType name="Hello_PortType">
    <operation name="sayHello">
        <input message="tns:SayHelloRequest"/>
        <output message="tns:SayHelloResponse"/>
    </operation>
</portType>
```

PortType operations

WSDL supports four basic patterns of operation

One-way

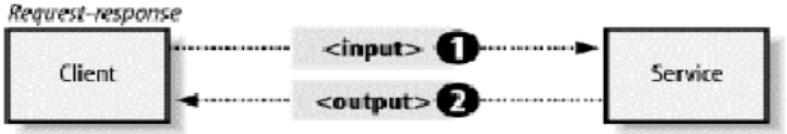
the service receives a messages. The operation therefore has a single input element.

Request-response

the service receives a message and sends a response.

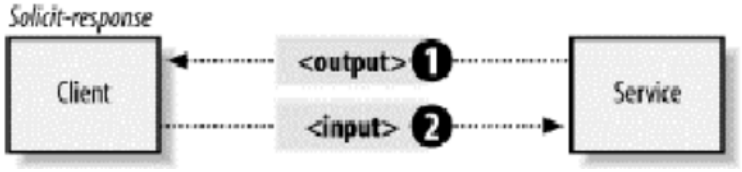
the operation therefore has one input element followed by one output element.

to encapsulate error fault element can also be specified.



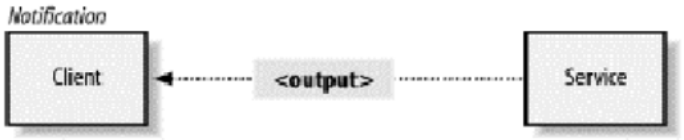
Solicit-Response

The service sends a message and receives a response. The operation therefore has one output element followed by one input element. To encapsulate error fault element can also be specified.



Notification

The service sends a message. The operation therefore has a single output element.



Binding element

- Binding element describes how the service will be implemented on the wire.
- WSDL includes built-in extensions for defining SOAP service and SOAP specific information.
- Binding element provides specific details on how a portType operation will actually be transmitted over the wire.
- Binding can be available via multiple transport HttpGet, HttpPost or SOAP.
- The binding element itself specifies *name* and *type* attributes.
- <binding name=“Hello-Binding” type=“tns:Hello-portType”>

  the type attribute references the portType

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
   <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
        <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:helloservice"
```

```
            use="encoded"/>
        </input>
    <output>
        <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:helloservice"
            use="encoded"/>
    </output>
</operation>
</binding>
```

- WSDL 1.1 includes built-in extension for soap1.1
- Specify SOAP specific details includes SOAP headers, SOAP encoding styles and SOAPAction http header.

Soap:binding

- This element indicates that the binding will be made available via internet.
- Style → overall style of soap message format.
- Style="rpc" specifies an rpc format.
    - Means the body of soap request will include wrapper xml element indicate function name.
    - Function parameters are then embedded inside the wrapper element
    - Likewise reponse will include a wrapper xml element that mirrors function request. Return values are then embedded inside the response wrapper element.
- A style value of document specifies an XML document call format.
- This means that the request and response messages will consist simply of XML documents.
- The transport attribute indicates the transport of the SOAP messages.
- http://schemas.xmlsoap.org/soap/http - indicates the SOAP HTTP transport,
- http://schemas.xmlsoap.org/soap/smtp - indicates the SOAP SMTP transport.

Soap:operation

- This element indicates the binding of a specified operation to a specific SOAP implementation.
- SoapAction -- > attribute specifies that soapAction http header be used for identifying the service.

Soap:body

- This element used to specify the details of input and output message.
- Specifies the soap encoding style and namespace URN associated with the specified service.

Service element

- Service element specifies the location of the service.

```
<service name="Hello_Service">
        <documentation>WSDL File for HelloService</documentation>
        <port binding="tns:Hello_Binding" name="Hello_Port">
                <soap:address
                        location="http://localhost:8080/soap/servlet/rpcrouter"/>
        </port>
</service>
```

- Because this is a soap service , we use soap:address element
- **<documentation>** element to provide human readable documentation.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<message name="SayHelloRequest">
<part name="firstName" type="xsd:string"/>
```

```
</message>
<message name="SayHelloResponse">
<part name="greeting" type="xsd:string"/>
</message>
<portType name="Hello_PortType">
<operation name="sayHello">
<input message="tns:SayHelloRequest"/>
<output message="tns:SayHelloResponse"/>
</operation>
</portType>
<binding name="Hello_Binding" type="tns:Hello_PortType">
<soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="sayHello">
<soap:operation soapAction="sayHello"/>
<input>
<soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:examples:helloservice"
use="encoded"/>
</input>
<output>
<soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:examples:helloservice"
use="encoded"/>
</output>
</operation>
</binding>
<service name="Hello_Service">
<documentation>WSDL File for HelloService</documentation>
<port binding="tns:Hello_Binding" name="Hello_Port">
<soap:address
location="http://localhost:8080/soap/servlet/rpcrouter"/>
```

</port>
</service>
</definitions>

1.4 SOAP

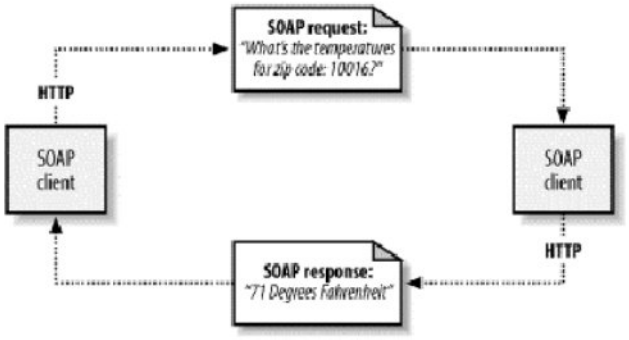
- Simple Object Access Protocol(SOAP)
- SOAP is an XML based protocol for exchanging information between computers
- Platform ,language independent.
- Initially SOAP is remote procedure call via http.
- Enable diverse applications easily exchange services and data.

SOAP specification defines three major parts:

- ***SOAP envelope specification***
    - defines specific rules for encapsulating data being transferred between computers.
    - This includes application-specific data, such as the method name to invoke, method parameters, or return values.
    - It can also include information about who should process the envelope contents and, in the event of failure, how to encode error messages.
- ***Data encoding rules***
    - own set of conventions for encoding data types.
- ***RPC conventions***
    - SOAP can use any messaging systems, including one-way and two-way messaging.
    - For two-way message- SOAP defines simple convention representing Remote procedure call(RPC)

SOAP conversation.

- The service method, getTemp , requires a zip code string and returns a single float value.



The SOAP Request

client request must include the name of the method to invoke and any required parameters.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
  <ns1:getTemp
    xmlns:ns1="urn:xmethods-Temperature"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <zipcode xsi:type="xsd:string">10016</zipcode>
  </ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

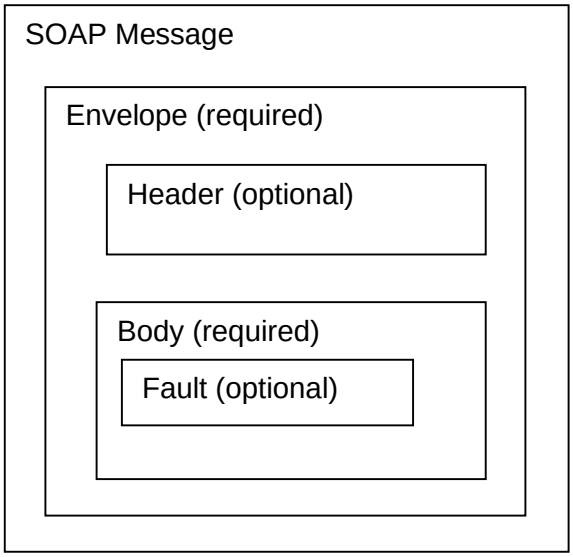
SOAP Response

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
      <ns1:getTempResponse
        xmlns:ns1="urn:xmethods-Temperature"
       SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <return xsi:type="xsd:float">71.0</return>
     </ns1:getTempResponse>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

SOAP Message Structure

- A one-way message, a request from a client, or a response from a server is officially referred to as a *SOAP message.*
- Every SOAP message has a mandatory *Envelope* element, an optional *Header* element, and a mandatory *Body* element.



SOAP – Envelope

- Every SOAP message has a root Envelope element.
- SOAP uses XML namespaces to differentiate versions.
- The version must be referenced within the Envelope element.

- SOAP 1.1 namespace URI is *http://schemas.xmlsoap.org/soap/envelope/,*
- SOAP 1.2 namespace URI is *http://www.w3.org/2001/09/soap-envelope*.

SOAP Header

- optional Header element offers additional application-level requirements.
- used to specify a digital signature for password-protected services
- it can be used to specify an account number for pay-per-use SOAP services.
- provides an open mechanism for authentication, transaction management, and payment authorization.
- two header attributes: Actor attribute, MustUnderstand attribute
- **Actor attribute**
  - the client can specify the recipient of the SOAP header.
- ***MustUnderstand attribute***
  - Indicates whether a Header element is optional or mandatory.
  - If set to true, the recipient must understand and process the Header attribute according to its defined semantics, or return a fault.

```
<SOAP-ENV:Header>

    <ns1:PaymentAccount xmlns:ns1="urn:ecerami" SOAP-ENV:
        mustUnderstand="true">

        orsenigo473

    </ns1:PaymentAccount >

</SOAP-ENV:Header>
```

SOAP – Body Element

- Body
  - Body element is mandatory for all SOAP messages.
  - Body element include RPC requests and responses.

SOAP- Fault Element

- In the event of an error, the Body element will include a Fault element.

Element name	Description
faultCode	A text code used to indicate a class of errors.
faultString	A human-readable explanation of the error.
faultActor	A text string indicating who caused the fault.
detail	An element used to carry application-specific error messages.

SOAP- Fault Element

Name	Description
SOAP-ENV:VersionMismatch	SOAP Envelope element included an invalid namespace,
SOAP-ENV:MustUnderstand	recipient is unable to properly process a Header
SOAP-ENV:Client	client request contained an error.(nonexistent method name,incorrect parameters to the method.)
SOAP-ENV:Server	server is unable to process the client request. (service providing product data may be unable toconnect to the database.)

SOAP- Fault Element

```
<?xml version='1.0' encoding='UTF-8'?>

<SOAP-ENV:Envelope

        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"

        xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
```

```
        xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
        <faultcode xsi:type="xsd:string">SOAP-ENV:Client</faultcode>
        <faultstring xsi:type="xsd:string">
                Failed to locate method (ValidateCreditCard) in class
                (examplesCreditCard) at /usr/local/ActivePerl-5.6/lib/
                site_perl/5.6.0/SOAP/Lite.pm line 1555.
        </faultstring>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```