Tema 5: CORBA

E. U. Informática en Segovia
Departamento de Informática
Universidad de Valladolid

5.1 Introducción

OMG (*Object Management Group*) (1989) consorcio de más de 700 empresas del sector informático que aboga por el uso de sistemas abiertos con interfaces estándar orientadas a objetos.

CORBA = *Common Object Request Broker Architecture* (arquitectura común de intermediarios de solicitudes a objetos).

Objetivo: sistema mediador entre cliente y métodos de un objeto. Las funciones del sistema: localizar el objeto, activarlo y pasarle la solicitud.

CORBA: estándar aprobado en 1991. En 1996 CORBA 2.0.

En CORBA todo está especificado en el IDL (*Lenguaje de Definición de Interfaz*) puramente declarativo. Definen puntos de entrada, argumentos y resultados de la llamada.

5.2. Invocación remota en CORBA

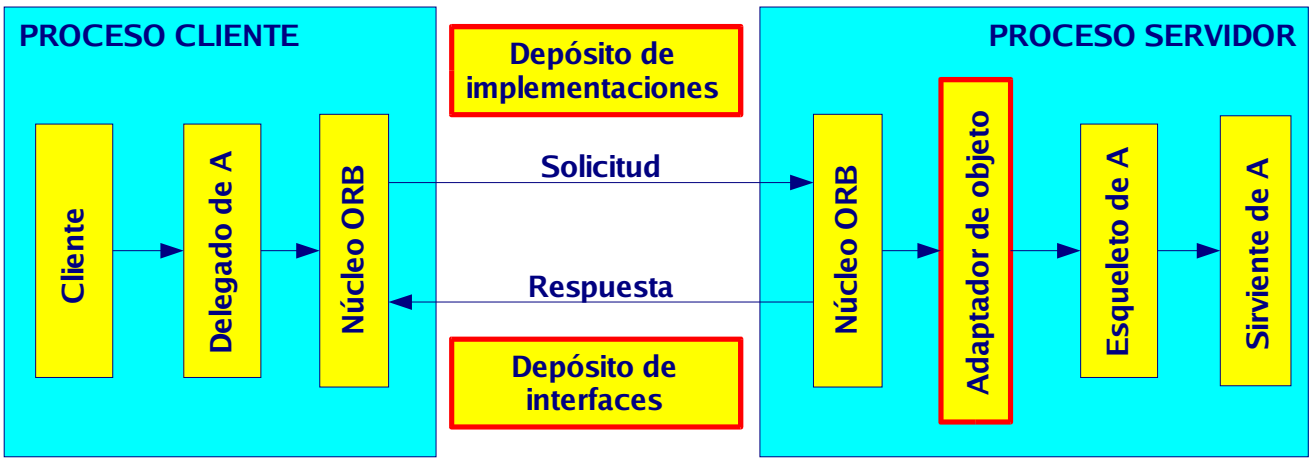
Similitudes con modelo general IMR:

- El programador define las interfaces remotas
- El compilador de interfaz genera delegados y esqueletos:
  - Delegados en el lenguaje del cliente
  - Esqueletos en el lenguaje del servidor

Diferencias con modelo general IMR:

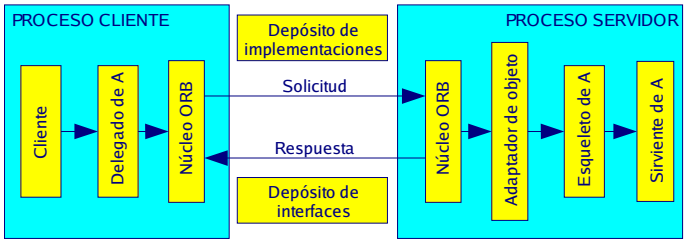
- Los clientes no tiene porqué ser objetos
- Objeto CORBA: objeto remoto que implementa interfaz remota y que puede estar escrito en cualquier lenguaje ⇒ CORBA no soporta la noción de clase.

5.2.1. Arquitectura de CORBA

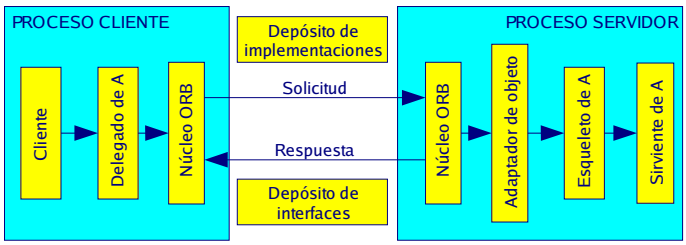


Tres componentes nuevos:

- Depósito de implementaciones
- Depósito de interfaces
- Adaptador de objetos



- Núcleo del ORB: módulo de comunicaciones. Interfaz con las siguientes funciones adicionales:
  - Funciones para arrancar y detener el núcleo del ORB.
  - Conversiones entre RORs y cadenas.
  - Operaciones que proveen las listas de argumentos necesarios para realizar invocaciones dinámicas.
- Adaptador de objetos: cubre el hueco entre los objetos CORBA (definidos en IDL) e interfaces de lenguajes de programación del servidor (tabla de correspondencias entre nombres de objetos y sirvientes).
- Delegados y esqueletos: interfaz para las llamadas, alineamiento y desalineamiento de argumentos...



- Depósito de implementaciones: registro de servidores (= adaptadores de objetos) usado bajo demanda. Es una tabla de correspondencias entre:
  - Nombres de adaptadores de objetos.
  - Caminos a implementaciones de objetos.
  - Nombres y puertos de las máquinas servidoras.
- Depósito de interfaces: da información sobre interfaces IDL registrados a clientes y servidores que lo requieran (nombres, métodos, argumentos y excepciones). Útil en caso de *invocación dinámica* (invocación en cliente sin delegado)

5.2.2. CORBA IDL

CORBA IDL permite definir módulos, interfaces, tipos, atributos y signaturas de métodos.

Consiste en un subconjunto de C++ con construcciones adicionales. Vemos algunas cosas en detalle

- Módulos: módulo = espacio de nombres. Por ejemplo, module Prueba en Java provocará la inclusión de la línea

  package Prueba

- Interfaces: definen los métodos disponibles en objetos CORBA que lo implementan. En Java, se traducen directamente en un interfaz Java.

- Métodos: por ejemplo:

  void getPerson(in string name, out Person p);

  - in: parámetro pasado cliente → objeto CORBA
  - out: parámetro pasado objeto CORBA → cliente
  - inout: parámetro pasado cliente ↔ objeto CORBA

  el resultado siempre out o void

Con oneway en la definición

oneway void llamada(in int num);

se cambia la semántica *al-menos-una-vez* (la que tiene por defecto) a semántica *tal-vez* ⇒ el cliente no se bloquea

Se pueden especificar excepciones que contengan variables. Por ejemplo:

```
exception DivisionPorCero{
   float op;
};
float div (in float nb1, in float nb2)
   raises (DivisionPorCero);
```

- Tipos: CORBA IDL soporta quince tipos primitivos: short, long, unsigned short, unsigned long, float, double, char, boolean, octet y any (representa cualquier tipo primitivo o construido).

  Constantes definidas usando la palabra clave const

  Tipo Object: se usa para RORs. Si un parámetro o resultado es del tipo Object, entonces se puede referir a cualquier objeto CORBA.

  Todos los argumentos o resultados pasados por valor, menos los del tipo Object, cuyo valor es una ROR.

  Tipos estructurados como uniones, registros, enumeraciones, matrices... Deben definirse con typedef antes de su uso

- Atributos: son como campos públicos de clases Java (variables de clase). Pueden ser readonly ⇒ valor no modificable

  Definición de atributo ⇒ dos métodos: uno para obtener el valor y otro para cambiarlo (solamente el primero si el atributo es readonly)

  readonly attribute long sum;

- Herencia: los interfaces IDL pueden ser extendidos. Ej: interface B:A{...};

  A añadirá nuevos tipos, constantes, excepciones, métodos y atributos a B

  La herencia múltiple es posible (interface C:A,B...). CORBA IDL no permite reutilizar nombres en caso de herencia múltiple

5.2.3. Servicios CORBA

Especificación de servicios utilizables por objetos distribuidos:

- Servicio de nombramiento (*naming service*): mantiene correspondencia nombres ↔ RORs de objetos CORBA con estructura jerárquica

  Se parte de un contexto de nombramiento inicial y cada nombre puede asociarse a una ROR o a otro contexto de nombramiento

- Servicio de eventos: define interfaces CORBA para notificar desde objetos de interés (*suministradores*) a objetos suscriptores (*consumidores*)

  Suministradores implementan interfaz PushConsumer con método push con argumento cualquier objeto CORBA.

  Consumidores implementan interfaz PullSupplier con método pull que devuelve un objeto CORBA como resultado

  Ambos registran sus ROR ⇒ canal de eventos (objeto CORBA) que permite comunicación asíncrona de múltiples suministradores y múltiples consumidores

- Servicio de notificación: extensión del servicio de eventos que añade:
  - Definir notificaciones como estructuras de datos
  - Filtros para consumidores (especificación de eventos de su interés)
  - Descubrimiento por suministradores de eventos de interés para consumidores
  - Descubrimiento por consumidores de eventos nuevos ofrecidos por suministradores
  - Configuración del canal, delegado o evento concreto: garantías de entrega, prioridades, ordenación de los eventos, políticas de descarte...
- Servicios de seguridad:
  - Servicios de autenticación
  - Servicios de control de acceso a los objetos CORBA
  - Servicios de auditoría de las IMRs por parte de los servidores
  - Servicios de no repudiación en los servidores

- Servicios de comercio (*trading service*): servicio de directorio similar al de nombramiento que permite localizar servicios por sus atributos
- Servicios de transacciones y de control de la concurrencia: el cliente especifica transacciones como secuencias de invocaciones remotas. Se pueden aplicar cerrojos de acceso a objetos CORBA (útil si un objeto está involucrado en una transacción)
- Servicios de objetos persistentes: los ORBs activan los objetos persistentes que están desactivados

5.3. Un ejemplo en Java

Ejemplo: el servidor va a mantener un índice que va a aumentar cada vez que el cliente realice una invocación.

Tareas a realizar:

1. *Definir el interfaz* y escribir el fichero IDL. *Compilar* el interfaz
2. *Implementar el cliente*
3. *Implementar el servidor*
4. *Ejecutar la aplicación*

1. Definir, escribir y compilar el interfaz IDL

Fichero count.idl:

```
module Counter
{
   interface Count
   {
      attribute long sum;
      long increment();
   };
};
```

Compilación:

```
$idlj -fall count.idl
```

Genera ficheros en el directorio Counter entre los cuales CountOperations.java:

```
package Counter;

/**
* Counter/CountOperations.java .
* Generated by the IDL-to-Java compiler (portable), version "3.1"
* from count.idl
* sábado 20 de noviembre de 2004 18H07' CET
*/

public interface CountOperations
{
  int sum ();
  void sum (int newSum);
  int increment ();
} // interface CountOperations
```

2. Implementación del cliente

```
/* CountClient.java */
import Counter.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CountClient {
  public static void main(String args[]) {
    try{
      ORB orb = ORB.init(args, null);

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

      NameComponent nc = new NameComponent("Count", " ");
      NameComponent path[] = {nc};
      Count countRef = CountHelper.narrow(ncRef.resolve(path));

      countRef.sum((int)0);
      for (int i = 0; i<1000; i++) {
         countRef.increment();
      }
      System.out.println("Suma: "+countRef.sum());

    } catch(Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
    }
  }
}
```

Inicialización del ORB local

Se obtiene el contexto de nombramiento

Se obtiene la ROR del servicio Count

Se usan los métodos igual que en local

Esto se compila:

```
$javac CountClient.java
```


3. Implementación del servidor

```
* CountServer.java */
import Counter.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

public class CountServer {
   public static void main(String args[]) {
      try{
         ORB orb = ORB.init(args, null);
         POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");
         rootpoa.the_POAManager().activate();

         CountServant countRef = new CountServant();
         countRef.setORB(orb);

         org.omg.CORBA.Object ref = rootpoa.servant_to_reference(countRef);
         Count cref = CountHelper.narrow(ref);

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

         NameComponent nc = new NameComponent("Count", " ");
         NameComponent path[] = {nc};
         ncRef.rebind(path, cref);

         System.out.println("Servidor preparado y esperando...");
         orb.run();
      } catch(Exception e) {
         System.err.println("ERROR: " + e);
         e.printStackTrace(System.out);
      }
   }
} .....
```

Inicialización del ORB local

Conexión del sirviente con el ORB local

Se esperan invocaciones

Se incluye la ROR en el contexto inicial de nombramiento

```
* CountServer.java (continuación)*/

....


class CountServant extends CountPOA {
   private ORB orb;

   public void setORB(ORB orb_val){
      orb = orb_val;
   }

   private int sum;

   public int sum() {
      return sum;
   }
   public void sum(int val) {
      sum = val;
   }
   public int increment() {
      sum++;
      return sum;
   }
}
```

Esto se compila:

```
$javac CountServer.java
```


4. Ejecución de la aplicación

Ejecución del servidor de nombres:

```
$tnameserv -ORBInitialPort 2000
Contexto de asignación de nombres inicial:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e3000000000000010000000000000007c000102000000000a3132372e302e
302e320007d000000035afabcb0000000206449425c0000000100000000000000010000000d544e
616d6553657276696365000000000000000400000000a000000000000001000000010000002000000
0000000100010000000205010001000100200001010900000000100010100
TransientNameServer: estableciendo puerto para referencias a objeto iniciales en
: 2000
Listo.
```

Ejecución del servidor:

```
$java CountServer -ORBInitialHost localhost -ORBInitialPort 2000
Servidor preparado y esperando...
```

Ejecución del cliente:

```
$java CountClient -ORBInitialHost localhost -ORBInitialPort 2000
Suma: 1000
```

5.4. Ejemplo con retrollamada

Ejemplo: el servidor va a mantener un índice que va a aumentar cada vez que el cliente realice una invocación. Cuando llegue a 500, el cliente informa al servidor mediante una *retrollamada*

Es invertir los papeles: el cliente se convierte en servidor

Tareas a realizar:

1. *Definir el interfaz* y escribir el fichero IDL. *Compilar* el interfaz
2. *Implementar el cliente*
3. *Implementar el servidor*
4. *Ejecutar la aplicación*

1. Definir, escribir y compilar el interfaz IDL

Fichero count.idl:

```
module Counter {
   interface CountCallback {
      void callback(in string message);
   };
   interface Count {
      attribute long sum;
      long increment(in CountCallback objRef);
   };
};
```

Compilación:

```
$idlj -fall count.idl
```

2. Implementación del cliente

```
/* CountClient.java */
import Counter.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

class CountCallbackServant extends CountCallbackPOA {
   private ORB orb;
   public void setORB(ORB orb_val) {
      orb = orb_val;
   }
   public void callback(String notification) {
      System.out.println(notification);
   }
}
......
```

El cliente define la clase CountCallbackServant, que antes no estaba. Esta clase contendrá la implementación de los métodos que puedan ser invocados desde el servidor.


```
/* CountClient.java (continuación)*/

......
public class CountClient {
   public static void main(String args[]) {
      try{
         ORB orb = ORB.init(args, null);
         POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         rootpoa.the_POAManager().activate();

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

         NameComponent nc = new NameComponent("Count", " ");
         NameComponent path[] = {nc};
         Count countRef = CountHelper.narrow(ncRef.resolve(path));

         CountCallbackServant countCallbackRef = new CountCallbackServant();
         countCallbackRef.setORB(orb);

         org.omg.CORBA.Object ref = rootpoa.servant_to_reference(countCallbackRef);
         CountCallback cref = CountCallbackHelper.narrow(ref);

         countRef.sum((int)0);
         for (int i = 0; i<1000; i++)
         {
            countRef.increment(cref);
         }
         System.out.println("Suma: "+countRef.sum());
      } catch(Exception e) {
         System.out.println("ERROR : " + e);
         e.printStackTrace(System.out);
      }
   }
}
```

3. Implementación del servidor

```
* CountServer.java (continuación)*/

....

class CountServant extends CountPOA
{
   private ORB orb;

   public void setORB(ORB orb_val){
      orb = orb_val;
   }

   private int sum;

   public int sum() {
      return sum;
   }
   public void sum(int val) {
      sum = val;
   }
   public int increment(CountCallback callobj) {
      sum++;
      if (sum == 500)
      {
         callobj.callback("Suma acumulada: 500");
      }
      return sum;
   }
}
```


4. Ejecución de la aplicación

Ejecución del servidor de nombres:

```
$tnameserv -ORBInitialPort 2000
Contexto de asignación de nombres inicial:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e3000000000000010000000000000007c000102000000000a3132372e302e
302e320007d000000035afabcb0000000206449425c0000000100000000000000010000000d544e
616d6553657276696365000000000000000400000000a0000000000000100000001000000200000
0000000100010000000205010001000100200001010900000000100010100
TransientNameServer: estableciendo puerto para referencias a objeto iniciales en
: 2000
Listo.
```

Ejecución del servidor:

```
$java CountServer -ORBInitialHost localhost -ORBInitialPort 2000
Servidor preparado y esperando...
```

Ejecución del cliente:

```
$java CountClient -ORBInitialHost localhost -ORBInitialPort 2000
Suma acumulada: 500
Suma: 1000
```

5.5. Ejemplo con objetos persistentes

Para lograr objetos persistentes el programador deberá guardarlos en disco y recuperarlos cuando sea necesario.

Es un buen ejemplo para ver cómo se hace y cómo se utilizan las excepciones en los objetos CORBA.

Tareas a realizar:

1. *Definir el interfaz* y escribir el fichero IDL. *Compilar* el interfaz
2. *Implementar el cliente*
3. *Implementar el servidor*
4. *Ejecutar la aplicación*

1. Definir, escribir y compilar el interfaz IDL

Fichero count.idl:

```
module Counter
{
   interface Count
   {
      exception ErrorFichero{};
      attribute long sum;
      long increment();
      void recuperaValor() raises (ErrorFichero);
      void almacenaValor() raises (ErrorFichero);
   };
};
```

Compilación:

```
$idlj -fall count.idl
```

2. Implementación del cliente

```
/* CountClient.java */
import Counter.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CountClient {
   public static void main(String args[]) {
      try{
         // Create and initialize the ORB
         ORB orb = ORB.init(args, null);

         // Get 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("Count", " ");
         NameComponent path[] = {nc};
         Count countRef = CountHelper.narrow(ncRef.resolve(path));

         // Call the Count server object and print results
         countRef.recuperaValor();
         for (int i = 0; i<1000; i++) {
            countRef.increment();
         }
         countRef.almacenaValor();
         System.out.println("Suma: "+countRef.sum());

      } catch(Exception e) {
         System.out.println("ERROR : " + e);
         e.printStackTrace(System.out);
      }
   }
```


3. Implementación del servidor

```
* CountServer.java (continuación)*/

....

class CountServant extends CountPOA {
   ...
   public void recuperaValor() throws Counter.CountPackage.ErrorFichero {
         try {
            BufferedReader br = new BufferedReader(new FileReader("dato.txt"));
            sum = (new Integer(br.readLine())).intValue();
            br.close();
         } catch (Exception e) {
            throw new Counter.CountPackage.ErrorFichero();
         }
      }

   public void almacenaValor() throws Counter.CountPackage.ErrorFichero {
         try {
            BufferedWriter bw =  new BufferedWriter(new FileWriter("dato.txt"));
            String cadena = (new Integer(sum)).toString();
            bw.write(cadena, 0, cadena.length());
            bw.write('\n');
            bw.close();
         } catch (Exception e) {
            throw new Counter.CountPackage.ErrorFichero();
         }
      }
}
```

recuperaValor y almacenaValor incorporan en su definición un throws y su contenido está incluído en un bucle try-catch

4. Ejecución de la aplicación

Ejecución del servidor de nombres:

```
$tnameserv -ORBInitialPort 2000
Contexto de asignación de nombres inicial:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e3000000000000010000000000000007c000102000000000a3132372e302e
302e320007d0000000035afabcb0000000206449425c0000000100000000000000010000000d544e
616d6553657276696365000000000000000400000000a0000000000000100000001000000200000
0000000100010000000205010001000100200001010900000000100010100
TransientNameServer: estableciendo puerto para referencias a objeto iniciales en
: 2000
Listo.
```

Ejecución del servidor:

```
$java CountServer -ORBInitialHost localhost -ORBInitialPort 2000
Servidor preparado y esperando...
```

Ejecución del cliente:

```
$java CountClient -ORBInitialHost localhost -ORBInitialPort 2000
Suma: 1000
$java CountClient -ORBInitialHost localhost -ORBInitialPort 2000
Suma: 2000
$java CountClient -ORBInitialHost localhost -ORBInitialPort 2000
Suma: 3000
```