UNIVERSIDAD CARLOS III DE MADRID

DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA.
DESARROLLO DE APLIACIONES DISTRIBUIDAS

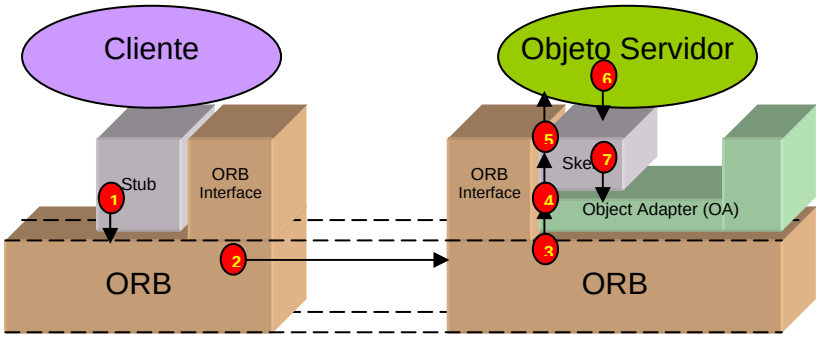
2 de septiembre de 2011

Para la realización del presente examen se dispondrá de **2 horas y media. NO** se podrán utilizar libros ni apuntes.

Ejercicio 1. (2.5 puntos)

Explica la arquitectura de CORBA: descripción de los elementos que conforman su estructura, flujo general de peticiones y características de las invocaciones en CORBA. ¿Qué es el Lenguaje de definición de interfaces (IDL)? ¿Qué ventajas tiene CORBA frente a Java RMI?

SOLUCIÓN:



1. **El cliente realiza una petición usando *stubs* estáticos (previamente compilados) y la dirige a su ORB.**
2. **El ORB cliente transmite las peticiones al ORB enlazado con el servidor.**
3. **El ORB del servidor redirige la petición al *adaptador de objetos* que ha creado el objeto destino.**
4. **El adaptador de objetos dirige la petición al servidor que implementa el objeto destino vía el skeleton.**
5. **El servidor devuelve su respuesta al skeleton el cual, devuelve el resultado siguiendo los mismos pasos.**

Explicar la funcionalidad de cada elemento del diagrama anterior.

- **IDL: Es un lenguaje que permite definir las interfaces de manera independiente del lenguaje de implementación.**
- **Existen proyecciones a diversos lenguajes de programación (mediante compiladores).**
- **Tiene tipos básicos, compuesto y mecanismos de herencia.**

Características de CORBA que lo diferencian de Java RMI:

- **Transparencia de la localización.**
- **Transparencia del servidor.**
- **Independencia del lenguaje.**
- **Independencia de la implementación.**
- **Independencia del nivel transporte.**

Ejercicio 2. (1.5 puntos)

Dado siguiente código de *.NET remoting*. Explica brevemente el significado de las líneas **destacadas** (únicamente L1, L2, L3, L4 y L5)

```
using System;
using System.Runtime.Remoting;

public class Listener{
   public static void Main(){
L1      RemotingConfiguration.Configure("Listener.exe.config");
      Console.ReadLine();
   }
}
```

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown
L2             mode="Singlecall"
L3             type="RemotableType, RemotableType"
L4             objectUri="RemotableType.rem"
             />
         </service>
         <channels>
L5          <channel ref="tcp" port="8989"/>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Solución:

L1: Indica el archivo de configuración del listener.
L2: Únicamente emplea un único objeto.
L3: Especifica que listener es un objeto remoto.
L4: Especifica la dirección del objeto remoto.
L5: Emplea protocolo http con el puerto 8989 en las comunicaciones.

Ejercicio 3. (1.5 puntos)

¿Qué protocolo de comunicación utiliza REST? ¿A través de qué representación se manipulan e identifican recursos con REST?
Se desea realizar una aplicación de REST sobre Facebook. Esta aplicación debe tener dos funcionalidades:

1. Una para acceder a los recursos del usuario, lista de amigos que han escrito en el muro, y lista de amigos que han comentado en las fotos.

2. Otra para crear un nuevo contenido en el muro de Facebook.

Indica qué métodos (asociados al protocolo de comunicación) emplearías para cada funcionalidad

Ejercicio 4. (2.5 puntos)

.

Dado el siguiente enunciado de la segunda parte de la práctica de Java RMI. Se pide:

- Dibujar la arquitectura resultante (indicando el rol de cada uno de los componentes). Indicar la distribución y contenido de los interfaces del servidor factoría, los objetos remotos y los clientes.
- Asumiendo el caso de un cliente que se suscribe al servicio de monitorización de CPU y recibe información cada 5 minutos. Se pide indicar qué se realiza internamente en cada una de las etapas de su funcionamiento (describir paso a paso).

Las arquitecturas de tipo cluster requieren de herramientas que permitan visualizar el estado de las máquinas. Se propone la realización de un sistema de monitorización de computadoras en un entorno distribuido, gracias a la librería de invocación de métodos remotos RMI de Java.

La característica que se estudiará en esta práctica será la cantidad de memoria y cpu en uso de la máquina, accesible mediante la lectura desde los ficheros /proc/meminfo y /proc/stat, respectivamente, y la cantidad de disco en uso (que consume el sistema de ficheros montado en el directorio raíz /) accesible mediante el comando df en los entornos UNIX.

El sistema distribuido consiste de tres aplicaciones:

1. Un servidor de factoría que contiene las referencias a los objetos remotos de monitorización y que se encarga de recibir las peticiones de los clientes.
2. Tres objetos remotos que monitorizan la cantidad de memoria, disco o CPU de la que se está haciendo uso en el sistema, y que se encargan de registrar las peticiones de los clientes (para poder enviarles los resultados de la monitorización).
3. Varios clientes, se registran en uno o varios servidores de monitorización a través del servidor de factoría y reciben los datos de dichas monitorizaciones (en porcentajes) cada cierto tiempo.

Desarrollo de la aplicación:

El servidor de factoría debe guardar las referencias de tres objetos remotos (es decir, no deben exportarse). Dichos objetos deben contener al menos un método:

- *suscribir* que recibe como argumento un objeto que implementa la interfaz remota de cliente y lo almacena para posteriormente enviarle información por callback. Así mismo, recibirá el intervalo de tiempo que determina cada cuánto se realizarán las monitorizaciones.

El servidor de factoría debe exportar un objeto remoto. Dicho objeto debe ofrecer al menos un método:

- *registrar* que recibe como argumento un objeto que implementa la interfaz remota de cliente. Así mismo, recibirá el intervalo de tiempo que determina cada cuánto se realizarán las monitorizaciones, y un indicador del servicio que se desee monitorizar (memoria, disco o cpu).

Todos los clientes deben contener al menos un método que será invocado por el servidor en el callback:

- *notificar* que recibe como argumento los datos de la monitorización del servidor.

Ejercicio 5. (2 puntos)

.

Dado el siguiente enunciado de la práctica de Corba. Se pide responder a las siguientes preguntas:

- Dibujar la arquitectura resultante para un cliente en Java y uno en C++.
- Indicar dónde se invoca el *callback*
- Indicar en qué paso de la ejecución se accede al servicio de nombrado tanto en los clientes como en el servidor
- ¿Qué pasa si se cae el servicio de nombres cuando la aplicación está ya funcionando?

Se trata de implementar un contador distribuido. La aplicación cliente/servidor estará formada por un servidor que contiene un objeto CORBA el valor actual del contador y varios clientes que se encargarán de incrementar el valor del contador cada cierto tiempo.

El servidor notificará a los clientes de cuando en cuando el valor del contador.

Los clientes obtendrán del servidor una referencia del objeto CORBA contador y a continuación lo usarán para incrementar el valor del mismo cada cierto tiempo.

Los procesos clientes solicitarán al usuario un número entero, el cual es necesario verificar que se encuentra correctamente escrito, y dicho valor será el intervalo de tiempo (en segundos) entre el que el cliente incrementará el valor del contador, una vez se haya obtenido éste a través del objeto CORBA que proporciona el servidor.

El proceso servidor solicitará al usuario un número entero, el cual es necesario verificar que se encuentra correctamente escrito, y dicho valor será el intervalo de tiempo (en segundos) entre el que el servidor notificará a los clientes (a través de callback) del valor del contador, una vez se haya obtenido un objeto CORBA que proporciona el cliente.

Con motivo de jugar con la interoperabilidad que es inherente a la especificación CORBA a partir de la versión 2, se habrán de implementar 2 objetos CORBA y sus correspondientes procesos clientes tanto en Java como en C++. El servidor únicamente tendrá que implementarse en Java.