Invocación remota y objetos distribuidos

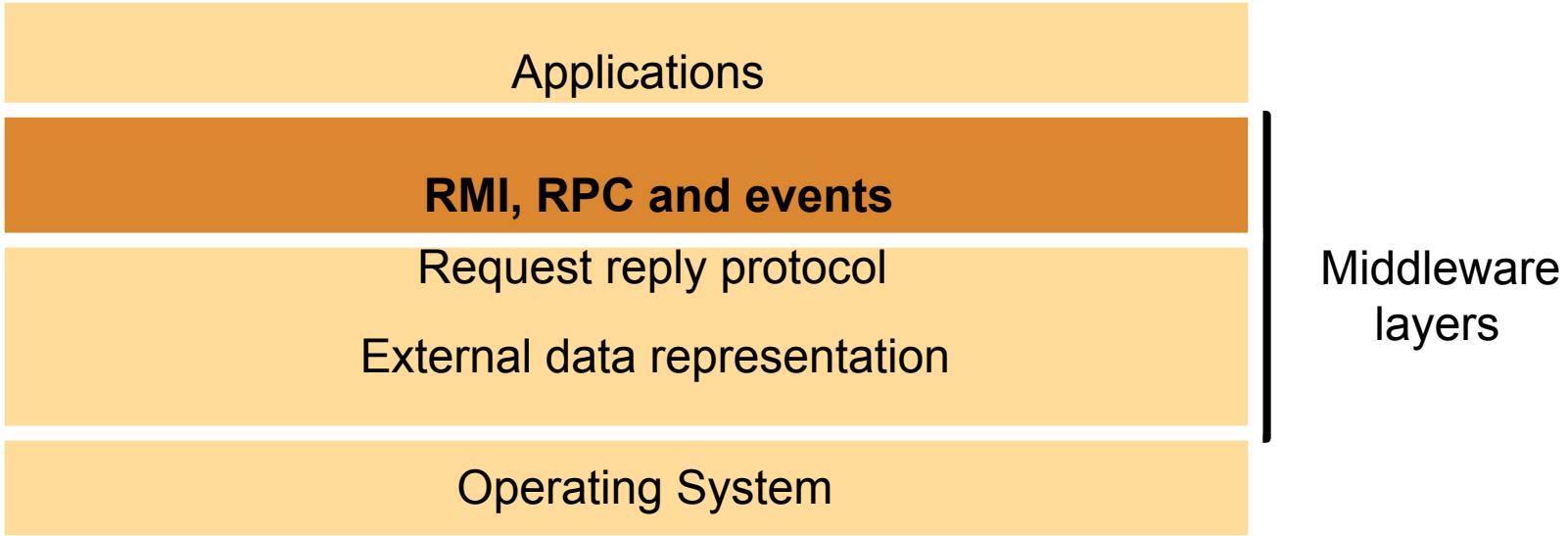
UNIVERSIDAD NEBRIJA

Índice

- Invocación a métodos remotos
- Llamada a un procedimiento remoto
- Eventos y notificaciones
- Java RMI

Invocación a métodos remotos

- Remote Method Invocation (RMI): El proceso invoca métodos de un objeto que reside en un nodo remoto



Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - Transparencia frente a la ubicación: el objeto que realiza la invocación NO puede distinguir si el objeto que invoca es local o no
  - Transparencia frente a los protocolos de comunicación: puede implementarse sobre UDP o TCP
  - Independencia del sistema operativo y del hardware: recuerda el empaquetado y desempaquetado de la información en puntos anteriores

Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - Interfaces de sistemas distribuidos: Basadas en el paso de parámetros de entrada y salida.

  NO PERMITEN EL ACCESO DIRECTO A VARIABLES, sólo mediante métodos

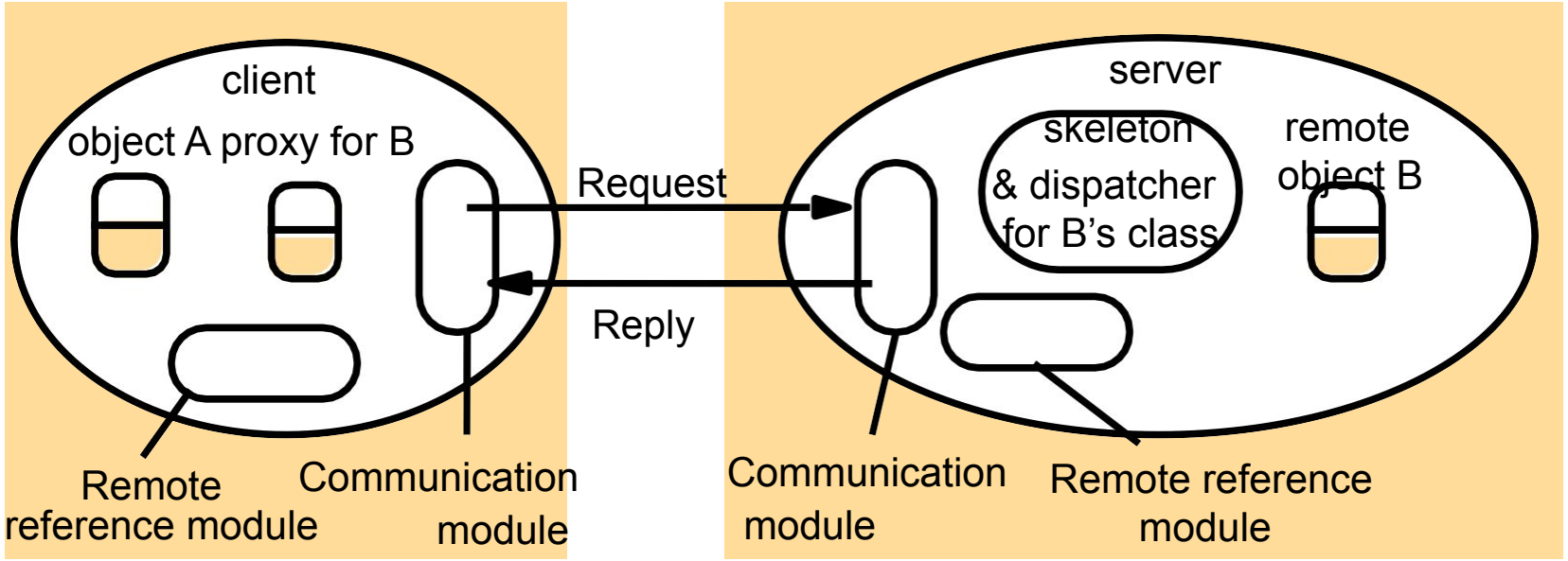
- En los sistemas distribuidos no se pueden intercambiar punteros
  - Interfaces de servicio: El servidor únicamente ofrece un número limitado de servicios al cliente. Remote procedure call (RPC)
  - Interfaces remotas: Basado en objetos. La interfaz especifica los métodos asociados a un objeto (con entradas y salidas), disponibles para su invocación

Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - Referencia a objeto remoto: identificador que puede usarse a lo largo de todo un sistema distribuido para referirse a un objeto remoto particular y único
  - Interfaces remotas: los objetos en otros procesos pueden invocar solamente los métodos que pertenezcan a su interfaz remota.
    - Las interfaces remotas NO PUEDEN incluir constructores, los objetos se crean en la sección de inicialización o mediante métodos
      Compactación automática de memoria: si el lenguaje soporta la compactación automática el sistema RMI debe permitir la compactación de objetos remotos
  - Excepciones: las invocaciones a objetos remotos deben ser capaces de lanzar excepciones debidas a fallos

Invocación a métodos remotos

- Remote Method Invocation (RMI):



Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - Módulo de referencia remota: Traduce las referencias entre objetos locales y remotos. Crea también referencias a objetos remotos
    - Tabla de objetos remotos almacena objetos remotos implementados y entradas de proxy. *Ejemplo*: objeto remoto B y proxy B
      - Cuando llega un objeto remoto por primera vez se crea la referencia y se añade a la tabla
      - Cuando llega una referencia a un objeto remoto dentro de un mensaje y no está dentro de la tabla el RMI crea un nuevo proxy y lo añade a la tabla

Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - Software RMI:
    - Proxy: asegura la TRANSPARENCIA de la invocación al método remoto.
      - Existe un proxy POR OBJETO remoto
    - Distribuidor: cada servidor tiene UNO POR CLASE. Recibe mensajes de petición y selecciona los métodos apropiados del esqueleto
    - Esqueleto: implementa los métodos de la interfaz. Proporciona los medios básicos para permitir que el middleware del servidor acceda a los objetos definidos por el usuario
      - El servidor tiene UNO POR CLASE

Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - Las clases para cada proxy, el distribuidor y el esqueleto se generan automáticamente mediante un compilador
  - Como la ubicación de los objetos puede variar con el tiempo, existen servicios de localización que ayudan a los clientes a encontrar objetos remotos desde sus referencias
  - Los sistemas distribuidos utilizan un compactador automático de memoria que asegura que mientras alguien posea una referencia al objeto remoto éste seguirá existiendo, pero cuando no haya ninguna referencia a él se recuperará la memoria que ocupa

Invocación a métodos remotos

- Remote Method Invocation (RMI):
  - En los objetos distribuidos el estado no es distribuido: reside en una sola máquina. Solamente las interfaces implementadas por el objeto están disponibles en otras máquinas
  - Invocación estática: utiliza definiciones de interfaz predefinidas.
    - Se conocen las interfaces de un objeto cuando la aplicación del cliente se está desarrollando
    - Si cambian las interfaces, entonces la aplicación del cliente debe ser recompilada antes de utilizar las interfaces nuevas
  - Invocación dinámica: a invocación a un método en tiempo de ejecución
    - Desarrollado para que soporte cualquier interfaz posible
    - Accede a los métodos de la interfaz solicitando un id del método remoto

Invocación a métodos remotos

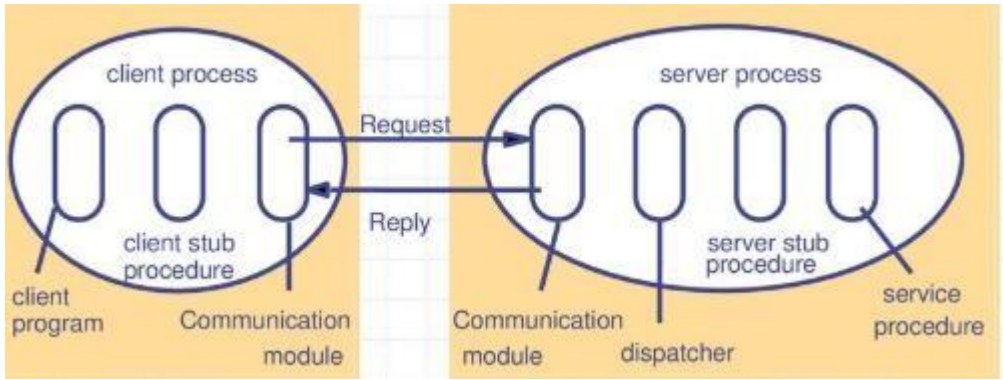
- Remote Method Invocation (RMI):
  - Seguridad en objetos remotos:
    - ESCENARIO: el desarrollador de un objeto remoto desarrolla también el proxy y lo registra mediante un servicio
    - PROBLEMAS:
      - Se proporciona un proxy falso y se secuestra el servicio
      - Cómo se autentica el servidor por parte del cliente si esta función depende del proxy
    - SOLUCIÓN:
      - El proxy se descarga de un servicio que permite al cliente verificar su origen
      - El proxy autentificará apropiadamente al objeto
      - El cliente deberá confiar en este comportamiento

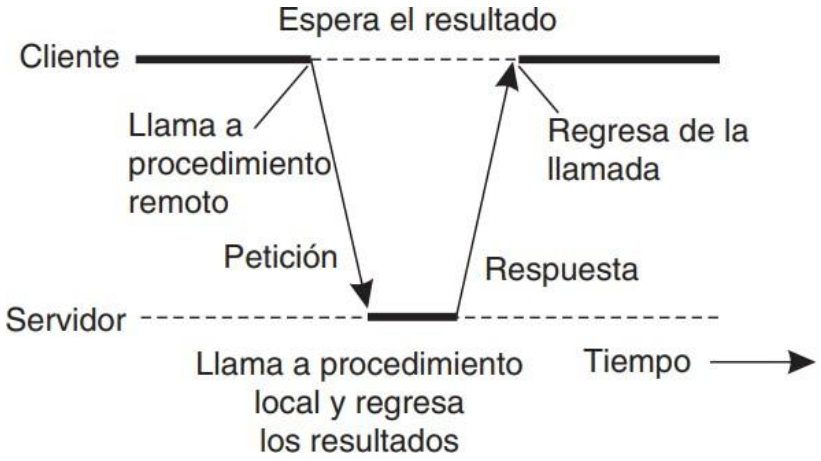
Ejercicio 30 minutos

- Implementa un Hola Mundo con RMI

Llamada a un procedimiento remoto

- Remote Procedure Call (RPC):





Llamada a un procedimiento remoto

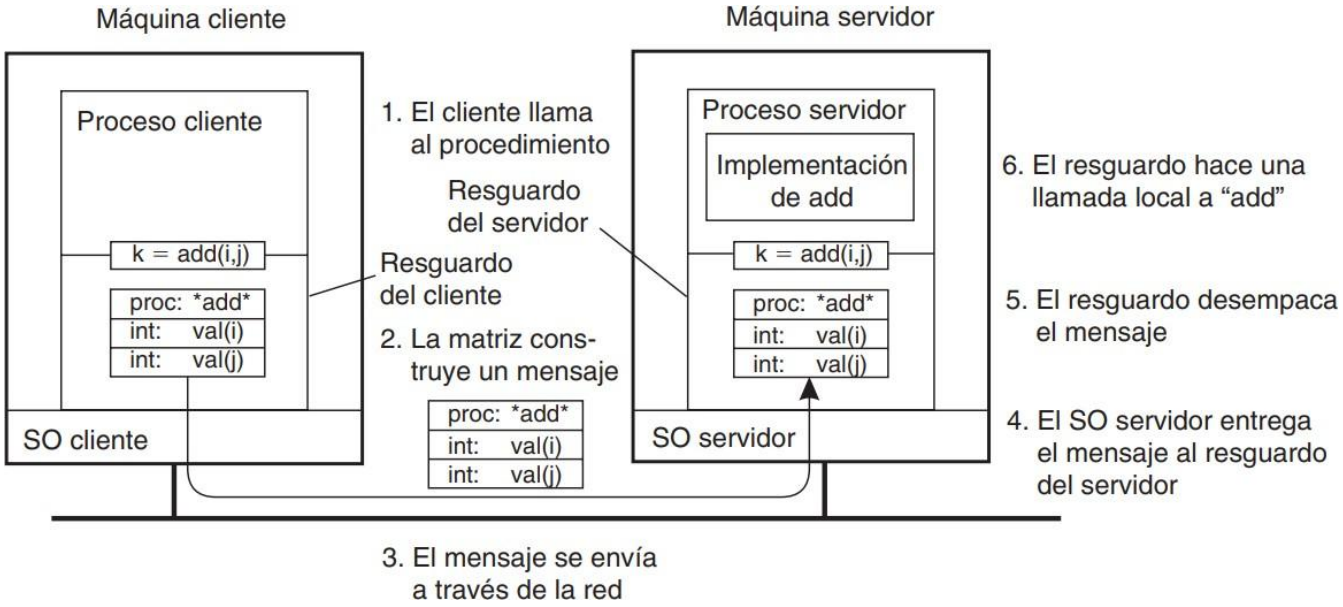
- Remote Procedure Call (RPC): Posee una implementación muy similar a la de RMI pero no incluye ninguna referencia a objetos remotos, ya que se basa en procedimientos.
  - Sustituye el proxy por procedimientos de resguardo para cada procedimiento definido en la interfaz
  - Procedimiento de resguardo: se comporta como un procedimiento local de cliente
    - Envía mediante el socket el identificador del procedimiento y los argumentos del mensaje de petición
  - Distribuidor: selecciona uno de los procedimientos de resguardo según el identificador y llama al procedimiento de servicio correspondiente (interfaz de servicio)

Llamada a un procedimiento remoto

- Remote Procedure Call (RPC):
  1. El procedimiento cliente llama al resguardo del cliente de manera normal
  2. El resguardo del cliente construye un mensaje y llama al OS local
  3. El OS del cliente envía el mensaje al OS remoto
  4. El OS remoto da el mensaje al resguardo del servidor
  5. El resguardo del servidor desempaqueta los parámetros y llama al servidor
  6. El servidor realiza el trabajo y devuelve el resultado al resguardo.
  7. El resguardo del servidor empaqueta el resultado en un mensaje y llama a su OS local.
  8. El OS del servidor envía el mensaje al OS del cliente.
  9. El OS del cliente da el mensaje al resguardo del cliente.
  10. El resguardo desempaqueta el resultado y lo regresa al cliente.

Llamada a un procedimiento remoto

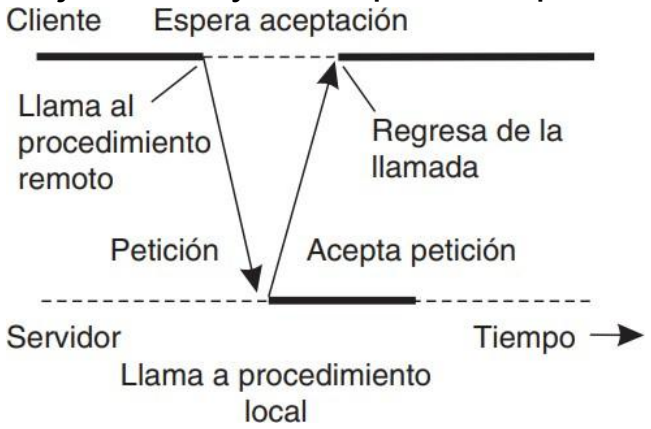
- Remote Procedure Call (RPC):



UNIVERSIDAD NEBRIJA

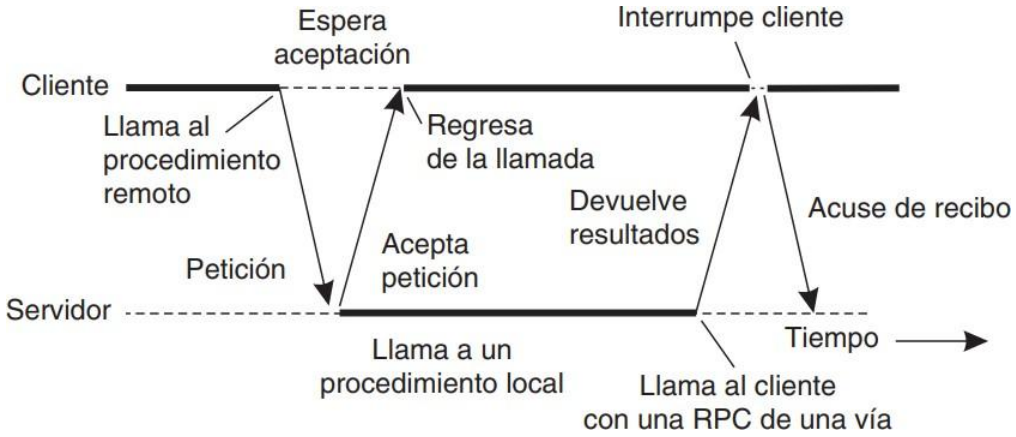
Llamada a un procedimiento remoto

- Remote Procedure Call (RPC):
  - RPC asíncronas: mediante las cuales, un cliente continúa trabajando de inmediato después de emitir la petición RPC
    - El cliente continuará su trabajo sin mayor bloqueo tan pronto reciba el acuse del servidor



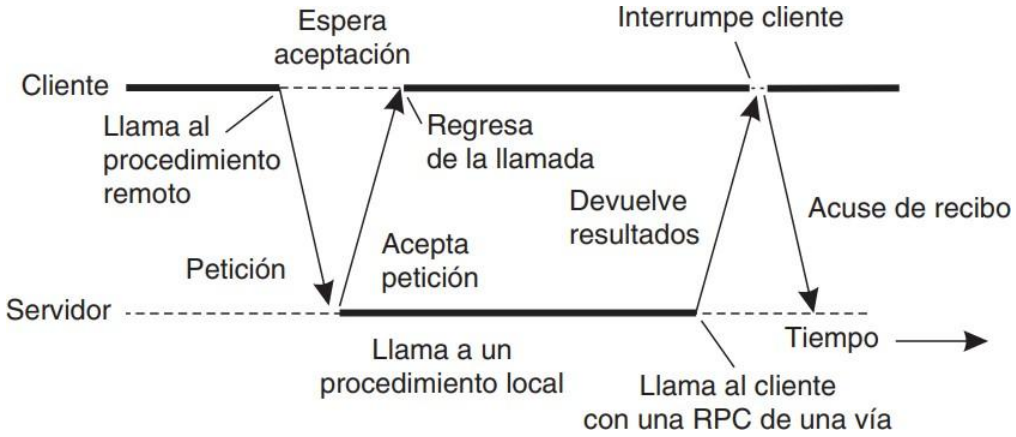
Llamada a un procedimiento remoto

- Remote Procedure Call (RPC):
  - RPC asíncronas: mediante las cuales, un cliente continúa trabajando de inmediato después de emitir la petición RPC



Llamada a un procedimiento remoto

- Remote Procedure Call (RPC):
  - RPC asíncronas: mediante las cuales, un cliente continúa trabajando de inmediato después de emitir la petición RPC



Eventos y notificaciones

- Eventos: utilizados para que un objeto pueda reaccionar a los cambios que ocurren en otro objeto
  - Heterogéneos: emplean notificaciones como mecanismo de comunicación entre objetos distribuidos.
    - Los objetos generadores de eventos publican los tipos de eventos que ofrecen y otros objetos se suscriben a los eventos y proporcionan una interfaz para recibir notificaciones
  - Asíncronos: los objetos generadores de eventos envían las notificaciones a todos los suscritos sin que sea necesarios que los suscritos se sincronicen

Eventos y notificaciones

- Participantes en una notificación de eventos distribuidos:
  - Objeto de interés: experimenta cambios de estado que pueden ser de interés para otros objetos
  - Evento: resultado de la finalización de un método
  - Notificación: información sobre el evento (tipo, atributos, identidad del objeto que lo genera, método que se invoca, *timestamp*, número de secuencia)
  - Suscriptor: objeto suscrito a los eventos de otro objeto
  - Objetos observadores: objeto mediador entre el objeto de interés y los suscriptores. Consulta al objeto de interés y posteriormente notifica a todos los suscriptores

Eventos y notificaciones

- Reglas para los observadores:
  - Encaminamiento: un observador encaminador puede llevar a cabo todo el trabajo de enviar las notificaciones a los suscriptores en representación de uno o más objetos de interés
  - Filtrado de notificaciones: un observador puede aplicar filtros para reducir el número de notificaciones recibidas, según el contenido de cada notificación
  - Patrones de eventos: busca relaciones entre varios eventos en posibles objetos de interés
  - Buzones de notificaciones: retrasa notificaciones si el suscriptor tiene un problema o fallo de comunicaciones

Java RMI

Consideraciones:

- En Java los objetos distribuidos se incorporaron al lenguaje
- Adopta objetos remotos como la única forma de objetos distribuidos
  - Un objeto remoto es un objeto distribuido cuyo estado siempre reside en una sola máquina y las interfaces son distribuidas
  - Para clonar un objeto remoto se necesita clonar el objeto en su servidor y también el proxy en cada cliente

Java RMI

Consideraciones:

INTERFAZ

- La interfaz debe importar siempre el paquete java.rmi
- La interfaz debe ser del tipo public interface
- El objeto que realiza la invocación debe saber que su destino es remoto para utilizar RemoteExceptions en todos los métodos
- El implementador del objeto remoto debe saber que es remoto para implementar la interfaz Remote
- A diferencia de los lenguajes multi-sistema como CORBA, el programador NO debe conocer cómo se relaciona el lenguaje con IDL. La interfaz es Java

Java RMI

Consideraciones:

- Existen dos clases:
  - Clase servidor: contiene la descripción del estado del objeto y una implementación de los métodos que operan en este estado
  - Clase cliente: contiene una implementación de un proxy. Se genera a partir de la especificación de la interfaz del objeto

Java RMI

Consideraciones:

- La clase debe ser Serializable. Recuerda que esta propiedad permite que el tipo de objeto sea transparente durante el intercambio de mensajes. Por tanto, CUALQUIER OBJETO SERIALIZABLE PUEDE PASARSE COMO ARGUMENTO O RESULTADO EN JAVA RMI.

- Todos los tipos de datos primitivos son serializables
- Los objetos locales se pasan por copia del valor
- Los objetos remotos se pasan por referencia

Java RMI

Consideraciones:

- Los objetos remotos se pasan por referencia
  - Una referencia a un objeto remoto se compone de la dirección de red y del puerto del servidor, así como también de un identificador local del objeto en el espacio de dirección del servidor
- El identificador local sólo lo utiliza el servidor, el resto de información la almacena el proxy como parte de su estado

Java RMI

Consideraciones:

- RMIregistry: es un enlazador de Java RMI que está presente en cada servidor que aloje objetos remotos. RMIregistry da soporte a una relación en forma de TABLA textual que conecta los nombres y las referencias de los objetos remotos
- Se requiere la clase Naming (con argumentos de string) para acceder a la tabla: //nombreEquipo:puerto/nombreObjeto
- nombreEquipo es la ubicación de RMIregistry. Si no se incluye es el localhost

Java RMI

Consideraciones:

TABLA

- La interfaz del RMIregistry implementa las siguientes funciones:
  - rebind: registra el identificador de un objeto mediante su nombre
  - bind: equivalente al anterior pero si el nombre existe lanza una excepción
  - unbind: se destruye el enlace de un determinado objeto con su nombre
  - lookup: busca un objeto remoto mediante su nombre en el registro

- RECUERDA: Desde el punto de vista del cliente el registro es también remoto

- Si se desea que los objetos remotos duren tanto tiempo como el proceso en el que se crean es necesario que la clase extienda UnicastRemoteObject

Java RMI

Windows:

- Definir interfaz remota (import java.rmi, extends Remote, throws RemoteException)
- Definir los objetos que se van a transmitir (extends Serializable)
- Implementar interfaz remota (extends UnicastRemoteObject, SecurityManager, RMIregistry-rebind)
- Definir el programa cliente (SecurityManger, lookup, ejecución de métodos remotos)

Java RMI

Windows:

- Compilaciones:
  - Compilar las interfaces remotas
  - Compilar la clase del servidor
  - Compilar la clase del cliente
  - Generamos un jar de todas las clases del paquete (interfaces y objetos serializables)
    jar cvf Nombre_del_paquete.jar Carpeta_del_paquete*.class
  - Para todos los que extienden UnicastRemoteObject aplicamos el siguiente comando (SÓLO PARA LAS IMPLEMENTACIONES DE LA INTERFAZ, NO PARA LA INTERFAZ)
    rmic paquete.nombre_de_la_clase

Java RMI

Windows:

- Compilaciones:
  - Nos aseguramos de que se genera la clase Stub para cada una de las implementaciones (Este paso y el anterior no son necesarios desde Java2 v5.0, las versiones posteriores soportan generación dinámica de stubs y los skeleton desaparecieron en la versión 1.2 de Java)
  - Arrancamos el servicio rmiregistry
  - Al instanciar al método SecurityManager, es imprescindible definir un fichero que genere políticas de seguridad (obligatorio a partir de Java2). El fichero se llama java.policy

java –Djava.security.policy =“Ruta”\java.policy NombreDelPaquete.NombreDelServidor

- Para el cliente ejecutamos un comando similar, añadiendo los argumentos del mismo. java –Djava.security.policy =“Ruta”\java.policy NombreDelPaquete.NombreDelCliente “ Argumento_0” “Argumento_1” … “Argumento_N”

CORBA

- Nuevos conceptos con respecto a Java RMI:
  - Modelo de **objeto de CORBA**:
    - Los **clientes son objetos** de manera obligatoria
    - Los clientes podrán ser cualquier programa que **envíe mensajes de remotos** y que pueda recibirlos
    - Un **objeto CORBA** es aquel objeto remoto que **implementa una interfaz IDL**, una **referencia a un objeto** remoto y **responde invocaciones**
    - Permite **implementar objetos** CORBA en **lenguajes no orientados a objetos**
      - EN CORBA NO EXISTEN CLASES
      - NO SE PUEDEN PASAR INSTANCIAS DE CLASES COMO ARGUMENTOS
      - SÍ QUE SE ADMITEN ESTRUCTURAS DE DATOS DE OTROS TIPOS Y COMPLEJIDAD

CORBA

- Nuevos conceptos con respecto a Java RMI:
  - **CORBA IDL**:
    - Especifica un **conjunto de métodos** que pueden utilizar los clientes
    - Los parámetros deben estar **marcados como entrada o s**alida (in/out/inout)
      - Si no hay parámetro se indica como void
    - El paso de objetos CORBA se realiza **por referencia**
    - El paso de **tipos primitivos** y construidos se copian y se pasan **por valor** (se realiza una copia del valor en el nodo de recepción)

  - **Servicio de nombres** de CORBA: Enlaza las **referencias a objetos CORBA** con los servidores
    - El sistema es complejo ya que existe una **estructura jerárquica** para cada uno de los nombres de objetos. Esto fuerza a que sea necesario **buscar una ruta**

CORBA

- Similar a una **invocación de método asíncrona RPC** porque el **cliente no se bloquea** con las peticiones

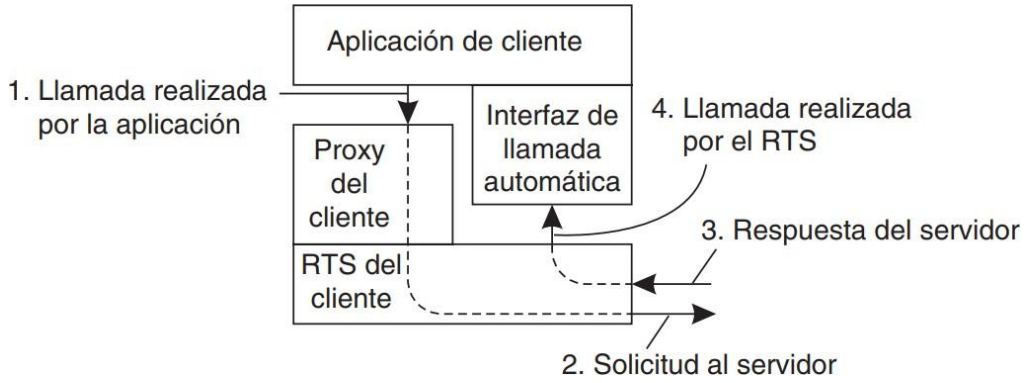
- Se realiza en dos pasos:

  - PASO 1: La interfaz original que implementa el objeto es reemplazada por dos nuevas interfaces que deben ser implementadas únicamente por el software del cliente
    1. Interfaz con la especificación de los métodos que el cliente puede invocar
    2. Interfaz automática: contiene un método que será invocado por el sistema en tiempo de ejecución del cliente para pasar los resultados del método asociado invocado por el cliente

```
void sendcb_add(in int i, in int j);          //Downcall realizada por el cliente
void replycb_add(in int ret_val, in int k);   //Upcall al cliente
```

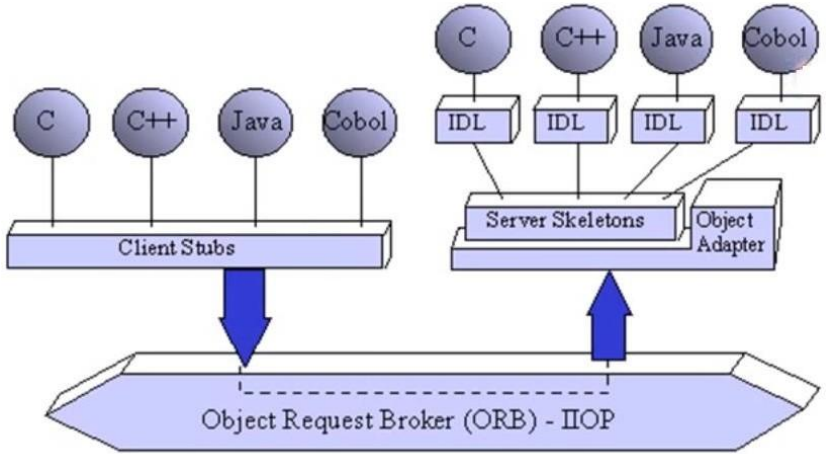
CORBA

- Similar a una invocación de método asíncrona RPC
- Se realiza en dos pasos:
  - PASO 2: Compilar las interfaces. Invocar los métodos en tiempo de ejecución (RTS)



CORBA

- Los mensajes enviados entre un cliente y un servidor son guardados por un sistema subyacente en el caso de que el cliente o el servidor ya no esté funcionando



CORBA

- Núcleo **ORB**: Tiene una interfaz mínima e incluye funciones para:
  - **Registrar y obtener referencias** a servicios iniciales (servicio de nombres) y arrancarla
  - **Intermediario** entre el esqueleto/stub y la red y el OS
- La interfaz **POA**: Portable Object Adapter (intenta aliviar el procesado del ORB)
  - El POA es la **conexión de un objeto con el ORB**
  - Un POA gestiona un **conjunto de objetos**
  - El POA gestiona las peticiones que llegan al servidor. Sus funciones incluyen:
    - **Gestionar los identificadores** de objetos
    - Recibir y gestionar **peticiones**
    - Establecer el modelo de **concurrencia**
    - Implementar las políticas de **objetos persistentes y transitorios**
  - Se denomina portable porque puede lanzar aplicaciones sobre cualquier ORB aunque esté producido por desarrolladores diferentes

CORBA

- **Esqueletos**: Es el **proxy del servidor**, se genera en el lenguaje del servidor (compilado IDL)
  - Desempaqueta los argumentos de las peticiones
  - Empaqueta los resultados y las excepciones

- **Stub**: Es el **proxy del cliente**, se genera en el lenguaje del cliente
  - Incluye los procedimientos de resguardo
  - Se genera desde la interfaz IDL y con el compilador de IDL
  - Desempaqueta los argumentos de las peticiones
  - Empaqueta los resultados y las excepciones

- El **esqueleto** y el **stub** son los encargados de **resolver las diferencias** entre OS, red y lenguaje de programación

CORBA

- **Módulos IDL**: Agrupa en unidades lógicas las interfaces y otras definiciones

```
module HelloApp
{
  interface Hello
  {
  string sayHello();
  oneway void shutdown(); // obligatorio incluirlo en todas las interfaces
                          // oneway indica que no espera respuesta del
                          // cliente

  };
};
```

CORBA

- **Interfaces IDL**: Interfaces lógicas
  - Soporta los siguientes lenguajes: C, C++, Java, COBOL, Smalltak, Ada, List, Pythone IDLScript
  - Cada uno de los lenguajes tiene una traducción estándar a CORBA IDL
    - Un **compilador generará automáticamente los proxies** necesarios (esqueletos y stubs) tanto para el cliente como para el servidor

```
package HelloApp;
public interface HelloOperations
{
  String sayHello ();
  void Shutdown ();
} // interface HelloOperations
```

CORBA

- **Interfaces IDL**: Interfaces lógicas
  - Soporta los siguientes lenguajes: C, C++, Java, COBOL, Smalltak, Ada, List, Pythone IDLScript
  - Cada uno de los lenguajes tiene una traducción estándar a CORBA IDL
    - Un **compilador generará automáticamente los proxies** necesarios (esqueletos y stubs) tanto para el cliente como para el servidor

      Hello.java //extends HelloOperations
      Hello**Helper**.java // **funcionalidades auxiliares** y avanzadas de CORBA
      Hello**Holder**.java // define los **parámetros del tipo out o inout** en la sintaxis
      // del lenguaje utilizado, en este caso JAVA
      _Hello**Stub**.java // **proxy del cliente** que interactúa con él
      Hello**POA**.java // OJO: implementa **POA** y **esqueleto** (**proxy del servidor**)

CORBA

- **Clases a implementar** en el servidor (requieren programación)
  - **Servant**: Define el funcionamiento de cada uno de los métodos declarados en la interfaz IDL

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloImpl extends _HelloPOA{

  private ORB orb;

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

  public String sayHello(){
    return "\nHello world !!\n";
  }

  public void shutdown(){
    orb.shutdown(false);
  }
```

CORBA

- **Clases a implementar** en el servidor (requieren programación)
  - **Servidor**: (main)

```
public class HelloServer {

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

      // get reference to rootpoa & activate the POAManager
      POA rootpoa =
      POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();

      // create servant and register it with the
      ORB  HelloImpl helloImpl = new HelloImpl();
      helloImpl.setORB(orb);

....
```

CORBA

Clases a implementar en el servidor (requieren programación)

- **Servidor**: (main)

```
……
 // get object reference from the servant
 org.omg.CORBA.Object ref =
 rootpoa.servant_to_reference(helloImpl);  Hello href =
 HelloHelper.narrow(ref);

 // get the root naming context
 // NameService invokes the name service
 org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
 // Use NamingContextExt which is part of the Interoperable
 // Naming Service (INS) specification.
 NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

 // bind the Object Reference in Naming
 String name = "Hello";
 NameComponent path[] = ncRef.to_name( name );
 ncRef.rebind(path, href);
……
```

CORBA

Clases a implementar en el servidor (requieren programación)

- **Servidor**: (main)

```
    ……

     System.out.println("HelloServer ready and waiting ...");

     // wait for invocations from clients
     orb.run();
    }

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

     System.out.println("HelloServer Exiting ...");

  }
}
```

CORBA

Clases a implementar en el cliente (requieren programación)

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloClient
{
  static Hello helloImpl;

  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");
        // Use NamingContextExt instead of NamingContext. This is part of the Interoperable naming Service.
        NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

        ……..
```

CORBA

Clases a implementar en el cliente (requieren programación)

```
……..
    // resolve the Object Reference in Naming
    String name = "Hello";
    helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

    System.out.println("Obtained a handle on server object: " + helloImpl);
    System.out.println(helloImpl.sayHello());
    helloImpl.shutdown();

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

CORBA

- **Protocolos inter ORB**
  - Gestionan la **capa de transporte**: qué se necesita para conectarse y liberar la conexión
  - Gestionan la **representación de datos** a un formato común: mediante marshalling y unmarshalling para cada tipo de datos del IDL
  - Gestiona el **tipo de mensajes** que se intercambia entre los diferentes clientes
- La capa que está por debajo del ORB se considera **un bus de objetos**

CORBA

- **Protocolos inter ORB**

  - Registra las referencia a objeto que está **codificada** de la siguiente forma:
    - **Tipo de objeto** CORBA
    - **Equipo** donde se localiza el objeto
    - Número de **puerto** del servidor del objeto
    - Cadena de caracteres que representa la **ID del objeto**

  - **Servicio de nombres**: es similar al rmiregistry (pero independiente del lenguaje de programación)
    - Incluye un formato basado en **URL** en el que se indica la **IP** o la URL del servidor y el puerto en el que el objeto está disponible, complicando más el sistema pero dotándolo de mayor flexibilidad

CORBA

- Entre los servicios más importantes proporcionados por CORBA encontramos:
  - Servicios de control de **concurrencia**
  - Servicio de **sincronización** de eventos
  - Servicio de **logging** de eventos
  - Servicio de nombres
  - Servicio de **planificación** de eventos (ordenación de eventos)
  - Servicio de gestión de **seguridad**
  - Servicio de marcas **temporales** (con relojes y estimación del error cometido)
  - Servicio de **transaciones**

CORBA

- **CORBA** : OMG (Object Management Group)
  - Pasos para un proyecto CORBA: creamos un **fichero IDL** (interface definition language)
  - **IMPORTANTE**: No llamar a la interfaz como al módulo
  - Compilar utilizando el comando idlj –fall xxxx.idl (Compilador en Java)
    - - fall genera los fichero stub, POA, Operations
  - Tras la compilación se creará **una carpeta con los ficheros Java** correspondientes a la interfaz
  - Compilar **TODOS** los archivos **Java generados**
  - Crear una **clase Servidor** que incluya: import xxxx.*; y que **extienda el POA** (Portable Object Adapter) generado tras la compilación del fichero idl
  - Implementar los **métodos definidos** en el fichero idl
  - Una vez creado el código Java creamos **iniciamos el ORB** (Object Request Broker) con el comando: start orbd –ORBInitialPort “puerto”. **OJO:** Requiere **permisos de admin**.
- **Iniciamos el proceso del servidor y del cliente** con los comandos:
  start java xxxxServer -ORBInitialPort “puerto” -ORBInitialHost “dirección”
  java xxxxClient -ORBInitialPort “puerto” -ORBInitialHost “dirección”