Especificación

Un Ejemplo: Servicio matemático.

La especificación del servicio en RPC de Sun:

```
/* matematico.x */

enum divstat{
  DIV_OK = 0,     /* NO ERROR */
  DIV_NOK = 1     /* ERROR */
};

/* Resultado de la división */
union resultado switch (divstat status) {
case DIV_OK:
        int cociente;
default:
        void;
};

struct enteros {
  int dividendo;
  int divisor;
};

program MAT_PROGRAM {
  version MAT_VERSION {
    resultado DIVIDIR(enteros)=1;
  }=1;
}= 0x03000000;
```

Los errores del servidor se recogen en variables

Se admite sólo:

- Un parámetro de entrada
- Un parámetro de salida

Se especifica:

- Nº de programa
- Nº de versión

La especificación en CORBA con idl:

```
// Matematico.idl

interface matematico {

  double dividir (in double  dividendo, in double divisor)

    raises(DivisionPorCero);

};
```

Se admite más de un parámetro de entrada y/o salida

Los errores del servidor se asocian a excepciones

Para ayudarnos a entender mejor lo que es CORBA, vamos a desarrollar un ejemplo sencillo en el entorno CORBA e irlo comparando con el clásico modelo de las RPC de Sun. Vamos a construir servicio matemático con una única operación que divide dos números enteros un.

Así, en la comparación con RPC comenzaremos a ver la especificación del servicio, construiremos un cliente que invoque un procedimiento remoto y, por último, construiremos un servidor que ofrezca el servicio.

En RPC de Sun, la especificación del servicio matemático se realiza con el lenguaje de especificación XDRL, y se recoge en el fichero `matematico.x`.

En el entorno CORBA, la especificación del servicio también se realiza en un lenguaje de especificación de interfaces, conocido como IDL. El fichero `Matematico.idl` muestra el servicio.

En el caso de las RPC, nos encontramos con que el diseñador debe asociar un número de programa y de versión al servicio matemático, así como numerar en orden ascendente las operaciones que ofrece la interfaz. (¿Qué ocurriría si dos interfaces distintos tuvieran el mismo número de programa y de versión?).

En CORBA no se indica un número de programa y de versión, sino simplemente el nombre del servicio.

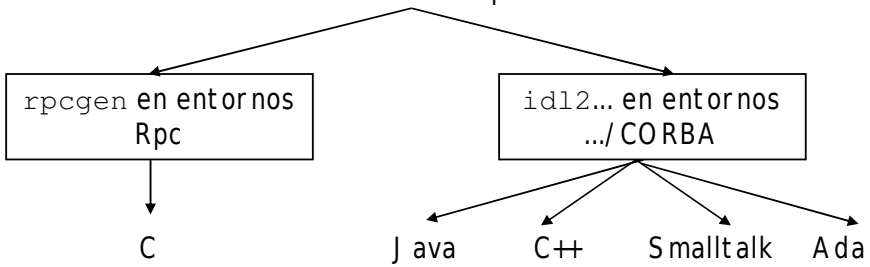
En RPC no se indica explícitamente si los parámetros son de entrada y/o salida. En CORBA, sin embargo, se puede indicar si son de entrada (`in`), de salida (`out`) o de entrada/salida (`inout`).

Al ejecutar la operación de división, el servidor puede devolver un resultado válido, o bien, un error de ejecución cuando se intenta dividir por cero.

Para poder recoger estas situaciones, en las RPC se construye un registro variante cuya variable `status` indicará con el valor `DIV_OK` si todo ha ido bien y, por tanto, `cociente` recoge el resultado, o bien, `DIV_NOK` para indicar que se ha producido un error de ejecución. Por tanto, el cliente deberá comprobar explícitamente este valor antes de recoger el cociente.

Para controlar situaciones de error en CORBA, se pueden declarar excepciones, como la que figura en el ejemplo: `DivisionPorCero` a la que se le podría asociar una determinada acción, por ejemplo, escribir un mensaje de error. También, se pueden producir excepciones predefinidas por CORBA como aquellas asociadas a errores de comunicación entre el cliente y servidor.

...Especificación



- En RPC el compilador `rpcgen` genera código C
- En CORBA hay múltiples compiladores `idl2c`, `idl2java`, ...
- En ambos entornos se generan *stubs* del cliente y servidor

Una vez definido el servicio, hay que someterlo al compilador de interfaces para generar los *stubs* del cliente así como el código de *dispacthing* del servidor.

En RPC, con el compilador `rpcgen matemático.x` se generan los ficheros de stubs en lenguaje C.

En CORBA existe también el concepto de compilación de interfaces, pero los lenguajes de programación a los que se traduce pueden ser múltiples: Java, Ada, C++, Smalltalk, etc.

Con el compilador `idl2java matematico.idl` se generan, al igual que RPC, los *stubs* del cliente y el código de *dispacthing* para el servidor en Java.

No vamos a enumerar la serie de ficheros que se crean porque depende del lenguaje de programación, por lo que solo iremos destacando aquellos más representativos.

El Cliente

Un Ejemplo en RPC y CORBA
El Cliente

Programación del cliente

1. Inicializa el soporte de transmisión
2. Localiza el servicio de nombres
3. Localiza el servicio matemático
4. Invoca el procedimiento de división
5. Analiza el resultado

Sistemas Distribuidos La Comunicación - 58

Programación de un cliente

Habitualmente, un cliente realiza las siguientes acciones:

1. Inicialización del soporte de transmisión.
2. Localización del servicio de nombres.
3. Localización del servicio matemático.
4. Invocación del método del servicio.
5. Utilización del resultado.

Además, dentro de estas acciones, hay que controlar que no se produzcan errores, ya sean de comunicación o de ejecución dentro del servidor.

Veamos a continuación cómo se realizan estas operaciones en los clientes de RPC y de CORBA respectivamente.

...El Cliente RPC

```
/* Cliente.c */
#include ...

main (int argc, char *argv[]){
  CLIENT                  * servidor;
  resultado       * resul;
  struct operandos {int dividendo; int divisor;};
  struct operandos oper;

  /* 1. Se inicializa el soporte de transmisión */   (1)
  /* 3. Se obtiene el id. Del servicio matemático */  (3)
  servidor =
        clnt_create("quijote",MAT_PROGRAM,MAT_VERSION,"udp");
  if (servidor == NULL) {{/* error de comunicacion */
    clnt_pcreateerror("quijote");
    exit(1);
  }
  /* Se obtienen los operandos */
  oper.dividendo = atoi(argv[1]);
  oper.divisor = atoi(argv[2]);

  /* 4. Se invoca la operación de división */   (4)
  resul = dividir_1(&oper, servidor);
  if (resul==NULL){/* error de comunicacion */
    printf("fallo en comunicación \n");
    exit(-1);
  }

  if (resul->status == DIV_NOK){/* Error de ejecución */
    printf("Intento de división n por cero \n");
    exit(-1);
  }

 /* 5. Se imprime el resultado */   (5)
  printf("%i/%i = %i \n ",oper.dividendo, oper.divisor,
         resul->resultado_u.cociente);
  exit(0);
}
```


En la descripción de cada uno de pasos de la programación del cliente vamos a ir viendo las diferencias entre las RPC y CORBA. En la figura de esta transparencia se muestra el código completo del cliente RPC, en la transparencia siguiente se muestra el de CORBA.

1. Inicialización del soporte de transmisión

Lo primero que se necesita para que cliente y servidor se comuniquen es un soporte de transmisión que permita el intercambio de mensajes.

En el caso de CORBA, el *Object Request Broker* (ORB) se encarga de realizar dicho intercambio, proporcionando un soporte de transmisión bidireccional y fiable que asegura, por defecto, que la semántica de entrega de mensajes es del tipo *como mucho una*. Además, se encarga de todas las tareas de *marshalling* y *unmarshalling*.

En el paso 1 “Inicialización del ORB” del cliente CORBA se inicializa este soporte virtual de transmisión abstrayéndose de los detalles de implementación de dicho soporte.

Con las RPC no existe esta transparencia, puesto que en este primer paso el cliente establece el protocolo de comunicación UDP. Además, la semántica de entrega de mensajes por defecto es al menos una.

2. Localización de servicio de nombres

En cuanto a la localización de un determinado servicio, la filosofía que se ha seguido sobre el servicio de nombres es completamente distinta en los dos entornos.

En RPC, el espacio de nombres no es global, sino local a cada una de las máquinas que constituyen el sistema, por esta razón es necesario que el cliente sepa en qué máquina reside el servicio (falta de transparencia de ubicación). Esto tiene el inconveniente de que puede haber en, distintas máquinas, dos servicios con el mismo nombre, que no tienen porqué dar el mismo servicio.

Además, el servicio de nombres de cada máquina, el *portmapper,* se ubica de manera estática, ya que se ejecuta en un puerto fijo (el 111) de cada máquina. Por esta razón la operación 2 (localización del servicio de nombres) no es necesaria en el modelo de RPC, pues no existe transparencia de ubicación de servicio.

Sin embargo, en CORBA aparece un espacio de nombres único que es soportado por un servicio de nombres que presenta reubicación dinámica. Así, en tiempo de ejecución (en la operación 2), el cliente pregunta al ORB dónde está el servicio de nombres, de tal manera que si el servicio de nombres cambiara de ubicación durante la ejecución del cliente, éste podría seguir funcionando si vuelve a preguntar de nuevo dónde reside este servicio.

El Cliente CORBA

```
// Client.java
import org.omg.CosNaming.*;

public class Client {
public static void main(String[] args) {
  try{
    // 1.Inicialización del ORB                                    (1)
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

    // 2.Se Obtiene el identificador del servicio de nombres       (2)
    org.omg.CORBA.Object
          rootObj = orb.resolve_initial_references("NameService");
    NamingContextExt
          root = NamingContextExtHelper.narrow(rootObj);

    // 3.Se Obtiene el identificador del servicio matemático       (3)
    org.omg.CORBA.Object
          mgrObj = root.resolve(root.to_name("MAT_PROGRAM"));
    Matematico  servidor = MatematicoHelper.narrow(mgrObj);

    // Se obtienen los operandos
    int dividendo = Integer.parseInt(args[0]);
    int divisor = Integer.parseInt(args[1]);

    // 4. Se invoca la operación de división                       (4)
    int cociente = servidor.dividir(dividendo,divisor);

    // 5. Se imprime el resultado                                  (5)
    System.out.println(dividendo + "/" +divisor+" = "+ cociente);
  }

  // Se controlan las excepciones del servicio
  catch (MatematicoPackage.DivisionPorCero e) {
    e.printStackTrace();
  }

  // se controlan los errores de comunicación y del sistema
  catch (Exception e) {
    e.printStackTrace();
  }
}
```


3. Obtención del servicio matemático

En RPC, los pasos 1 y 3 (Obtención del identificador del servicio matemático), recogen dos operaciones:

- Creación de un *socket* de tipo UDP, por donde el cliente podrá enviar y recibir información del servidor.
- Obtención del identificador de comunicación del servicio matemático preguntándoselo al *portmapper* de la máquina “quijote” . Este identificador solo es válido mientras viva el cliente, de tal manera que si el cliente falla, al rearrancarlo, ese identificador ya no será válido, aunque el servidor siga vivo.

En CORBA, el identificador que se obtiene al invocar la operación `resolve` del servicio `MAT_PROGRAM` sobre el servicio de nombres (paso 3), es válido incluso aunque el cliente falle o finalice correctamente. Si el cliente guarda ese identificador puede usarlo en sucesivas ejecuciones.

4. Invocación del método del servicio

Con el identificador de servicio ya se está en disposición de llamar a la operación de división (paso 4 del ejemplo).

En RPC, esta operación `dividir_1` obliga, tanto al empaquetamiento de los parámetros de entrada en un registro, como a la inclusión del identificador de comunicación del servidor como parámetro. Es decir, no presenta transparencia de acceso, ya que una llamada local a un procedimiento no presenta estas restricciones de llamada.

En CORBA, la llamada a un procedimiento viene determinada por el lenguaje que se use; en el caso de Java, al estar orientado a objetos, la invocación sigue el modelo `objeto.metodo (parametros)` tanto si es una llamada local como si es una llamada remota.

Por tanto, vemos que `servidor.dividir (dividendo, divisor)` presenta transparencia de acceso en la llamada, puesto que ya sea el objeto remoto o local se invoca de la misma forma. En el lado del cliente, la llamada a este método provoca la ejecución del *stub* correspondiente, que se encarga de pasar al ORB la petición de envío al servidor. Al ser una llamada síncrona el *stub* espera la respuesta que le pasa el ORB y la devuelve al programa principal.

5. Resultado.

Por último, en este paso se muestra por la salida estándar el resultado de la división, si todo hay ido bien.

...Programa del Cliente

Gestión de errores de transmisión y de ejecución en el servidor
- En RPC se tratan explícitamente
- En CORBA se usan excepciones

En RPC se pregunta EN CADA LLAMADA remota si:

- ¿`NULL`? ¿Hay error de transmisión?
- ¿`Status != NO_OK`? ¿Hay error de ejecución?

En CORBA se establece UNA SOLA VEZ el control con:

- `Catch (Exception e)` ¿Hay error de transmisión? (Señalada por el ORB)
- `Catch (MatematicoPackage.DivisionPorCero e)` ¿Hay error de ejecución? (Excepción señalada por el servidor)

Pero ¿y si ha habido problemas?

Los problemas que pueden producirse son errores de ejecución en el servidor (por ejemplo división por cero) o errores en la comunicación. Vamos a ver como se tratan:

En RPC los errores deben tratarse explícitamente ya que el lenguaje C no soporta excepciones. En el código del ejemplo, las cajas sombreadas recogen el control de errores de transmisión para cada llamada remota.

La caja de línea discontinua encierra el código que controla el error de ejecución especificado en el servicio: la división por cero, preguntado por el status devuelto por el servicio.

En CORBA, los errores de ejecución se pueden controlar asociándolos a una excepción, supuesto que el lenguaje de programación soporte excepciones. Cuando se intenta dividir por cero, el servidor produce la excepción `DivisiónPorCero`, que el cliente trata en el código de la caja de línea discontinua, imprimiendo el tipo de error que se ha producido.

Los errores de transmisión también se asocian a excepciones, pero no es el servidor quién las produce sino el soporte de transmisión de CORBA, es decir el ORB. En el cliente, el tratamiento de estos errores se recoge en la caja sombreada, y solo hay que hacerlo una vez, independientemente del número de llamadas remotas que se hagan.

Un Ejemplo en RPC y CORBA ¿Qué es un objeto CORBA?

Un cliente en Java solicita operaciones sobre un objeto CORBA.

¿Cualquier objeto Java es un objeto CORBA?

¡¡ NO !!

Un **objeto CORBA** es un **OBJETO VIRTUAL**

Un objeto de un lenguaje de programación es real

¿Qué relación hay entre un objeto CORBA y un objeto real?

La idea es similar a la de memoria virtual:

En el modelo de memoria virtual:

El proceso genera una dirección virtual

Gestor de memoria + MMU

Acceso a una dirección física o real

En el modelo CORBA:

El cliente referencia un objeto CORBA

ORB + POA

Acceso a un objeto real (C++, Ada, C, Cobol, ...)

El servidor solo ofrece objetos CORBA

¡¡ TRANSPARENCIA DE ACCESO !!

En el ejemplo anterior hemos visto que en el entorno CORBA un cliente escrito en lenguaje Java invoca operaciones sobre un objeto CORBA.

Entonces ¿se puede decir que todo objeto Java es un objeto CORBA ?

La respuesta en que no y su justificación recoge la esencia del modelo CORBA:

La relación que existe entre un objeto CORBA y un objeto en cualquier lenguaje de programación es similar a la que existe entre una dirección virtual y una dirección física en un sistema de memoria virtual de un sistema operativo. En un ordenador convencional, un proceso genera direcciones virtuales, direcciones que no existen realmente en la memoria principal, pero que con la ayuda de software de gestión de memoria y de la MMU se traducen de manera transparente a direcciones reales de memoria principal.

En el modelo CORBA, **un objeto CORBA es un objeto virtual** es decir no tiene existencia por sí mismo si no hay detrás un objeto en un determinado lenguaje de programación que le soporte.

Es decir, los objetos CORBA que referencia un cliente son tan virtuales como lo son las direcciones que genera un proceso. Cuando se referencia un objeto CORBA habrá un conjunto de componentes del sistema que traducirán esta referencia a un objeto real implementado en un determinado lenguaje de programación, al igual que el sistema de gestión de memoria traduce las direcciones virtuales a direcciones reales o físicas.

En CORBA los elementos encargados de esta traducción son el bus virtual (ORB) y un elemento denominado **adaptador de objetos** o POA (*Portable Object Adapter*), del que hablaremos más adelante.

¿Qué ventaja se obtiene al hacer que un objeto CORBA sea virtual?

Esto permite que cliente y servidor puedan escribirse en diferentes lenguajes de programación: El cliente escrito en Java puede invocar operaciones sobre un objeto CORBA soportado por un objeto en C++ o Ada, o incluso por un conjunto de rutinas en C o Cobol. Es decir, le abstrae al cliente de la implementación del objeto, presentado transparencia de acceso, puesto que independientemente de si el objeto real está soportado en un lenguaje orientado a objetos o no, el acceso siempre es del tipo `objeto.metodo(parámetros)`.

Por tanto, los servidores CORBA ofrecen siempre servicios representados por objetos CORBA.

De esto, también se desprende que un objeto CORBA y los objetos que les soportan (en algún lenguaje de programación concreto) tienen vidas independientes, es decir, tienen operaciones de creación y destrucción propias y pueden crearse y destruirse de manera independiente.

En el apartado siguiente veremos cómo se construye un servidor en RPC y CORBA y veremos que aquí es donde se presentan las mayores diferencias conceptuales.

El Servidor RPC

El Servidor

1. Inicializa el soporte de transmisión
2. Localiza el servidor de nombres
3. Da de alta del servidor matemático
4. Espera de peticiones
5. Invoca el procedimiento de división
6. Envía la respuesta

El Servidor RPC

```
/* matematico_svc.c */
main()
{...
  /* 1. Se crea un socket udp */ (1)
  transp = svcudp_create(RPC_ANYSOCK);
  /* 3. Se da de alta el servicio de nombres */ (3)
  svc_register(transp, MAT_PROGRAM, MAT_VERSION,
               mat_program_1, IPPROTO_UDP))
 /* 4. Se cede el control al dispatcher mat_program_1 */ (4)
  svc_run();
}
```

```
/* dispatcher mat_program_1 */
static void mat_program_1(rqstp, transp)
  ...
  /* Se analiza la petición */
  switch (rqstp->rqproc) {...
    case DIVIDIR:/* 5. Llamar al procedimiento de division */
      result = dividir_1(&argument, rqstp);  (5)
    break;
  }
  ...
  /* 6. Se envia la respuesta al cliente */ (6)
  svc_sendreply(transp, xdr_result, result);
  return;
}
```


Hay que recordar que, previamente a la construcción del servidor, hay que someter el fichero con la especificación del servicio al compilador de interfaces, para así obtener los ficheros con los perfiles de las operaciones y el *dispatcher.*

Un proceso servidor realiza habitualmente los siguientes pasos:

1. Inicializa el soporte de transmisión
2. Localiza el servidor de nombres
3. Da de alta el servicio en el servidor de nombres
4. Espera peticiones para ese servicio
5. Llama al procedimiento de división
6. Envía la respuesta al cliente

A continuación vamos a ver cómo se realiza cada uno de estos pasos en RPC y CORBA. En esta figura se muestra el servidor completo RPC y en la transparencia siguiente se ofrece el de CORBA.

1. Inicialización del soporte de transmisión

En el entorno RPC se crea un *socket* de tipo UDP para el envío y recepción, mientras que en el servidor CORBA simplemente se crea un bus virtual ORB.

2. Localización del servidor de nombres

En RPC, ya que no hay un servidor de nombres global, no hay que localizarlo, sino que directamente hay que dirigirse al *portmapper* de la máquina en la que reside el servicio deseado. El puerto por el que escucha el *portmapper* está prefijado y es el 111.

En CORBA, como el servidor de nombres es global y puede estar en cualquier máquina (y puede escuchar por cualquier puerto) hay que localizarlo, cosa que se hace solicitándoselo al ORB.

3. Alta del servicio en el servicio de nombres

En el entorno RPC se da de alta el servicio matemático “MAT_PROGRAM” en el *portmapper* o servicio de nombres local donde reside el proceso servidor.

En el entorno CORBA, el servidor da de alta, en el servidor de nombres global, el objeto CORBA `MAT_PROGRAM` que ofrecerá el servicio `Matematico`.

El código que realiza las tres operaciones anteriores en el entorno RPC (excepto el paso 2, que en RPC no tiene sentido) es generado automáticamente por el compilador de interfaz (`rpcgen`), recogiéndose en la parte `main` del esqueleto `matematico_svc.c`. Sin embargo, en CORBA estas tres operaciones se deben programar explícitamente.

El Servidor CORBA

Para ofrecer un servicio, el servidor

1. Crea un objeto real (servant)
2. Crea un objeto Corba
3. Los asocia
4. Da de alta el objeto CORBA

```
// Server.java
import org.omg.PortableServer.*; import org.omg.CosNaming.*;

public class Server {
   public static void main(String[] args) {
    try {
      // 1. Inicialización del ORB                                    (1)
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

      // Obtención del identificador del POA raiz
      POA rootPOA =
      POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      // Activación del POA manager
      rootPOA.the_POAManager().activate();

      // Creación de una implementación del servicio (servant)
      MatematicoImpl managerServant = new MatematicoImpl();

      // Se crea el objeto CORBA y se asocia al servant
      org.omg.CORBA.Object CORBAObj=
                  rootPOA.servant_to_reference(managerServant);

      // 2. Localización del servicio de nombres                      (2)
      org.omg.CORBA.Object rootObj =
                  orb.resolve_initial_references("NameService");
      NamingContextExt root =
                  NamingContextExtHelper.narrow(rootObj);

      // 3. Alta del s. matematico en el servicio de nombres          (3)
      root.bind(root.to_name("MAT_PROGRAM"),CORBAObj);

      // 4. Espera de peticiones                                      (4)
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
   }
}
```


4. Espera de peticiones

La operación 4 cede control al *dispatcher* que se encargará de recoger la petición, llamar al procedimiento adecuado, recoger la respuesta y mandarla al cliente. Tanto en RPC como en CORBA, el *dispatcher* lo genera automáticamente el compilador de interfaces.

En el modelo RPC, el *dispatcher* es el procedimiento `mat_program_1` incluido en esqueleto `matematico_svc.c` y en CORBA el *dispatcher* también se incluye en el esqueleto que genera el compilador de interfaces (no vamos a comentar aquí dicho esqueleto para no complicar el ejemplo).

5. Invocación del procedimiento de división

Veamos cómo se invoca el procedimiento adecuado cuando llega una petición.

En los entornos RPC es simple: cuando le llega la petición de división al *dispatcher*, éste llama al procedimiento `dividir_1` que es el que realmente realiza la división.

En CORBA también es el *dispatcher* el encargado de llamar al procedimiento de división cuando se recibe una petición de tal operación. Sin embargo, para que esa petición llegue al *dispatcher* el servidor tiene que realizar una serie de acciones antes de ponerse a esperar a que lleguen peticiones. Veámoslas.

> Cuando hablábamos del cliente CORBA decíamos que en el exterior sólo veía objetos CORBA, es decir veía objetos virtuales.También comentábamos que tras un objeto virtual debe haber un objeto real en algún lenguaje concreto que lo soporte. Pero ¿quién y cómo se realiza esta asociación?
>
> El encargado de este trabajo es el servidor, que, en nuestro ejemplo en Java, debe crear el objeto que soporta el procedimiento que realmente divide. (En la jerga de CORBA, a este objeto se le denomina ***servant***). Después creará un objeto CORBA y, por último, tendrá que asociar este objeto CORBA al objeto Java anterior. Después de esta asociación, el objeto CORBA está listo para darlo de alta en el servidor de nombres y hacerlo visible al exterior.

Todas estas tareas se enmarcan en un rectángulo sombreado en el código del servidor CORBA y no aparecen en el servidor de RPC porque son específicas de CORBA.

La asociación entre un objeto CORBA y el objeto real que lo implementa se guarda en un componente del sistema CORBA denominado **Adaptador Portable de Objetos** o POA (*Portable Object Adapter*).

Pero todavía nos queda contestar a la pregunta que teníamos pendiente al principio de este punto:¿Cómo invoca el dispatcher el procedimiento de división?

La función del *dispatcher* en CORBA es la misma que en RPC, analiza la petición y llama al procedimiento adecuado del objeto correspondiente.

...Programa del Servidor

El procedimiento de servicio en RPC:

```
/* dividir.c */
#include ...#
resultado *dividir_1(enteros *p1, struct svc_req *p2){
  resultado *res;
  res = (resultado *)malloc(sizeof(struct resultado));
  if (p1->divisor == 0)
    res->status = DIV_NOK;
  else {
    res->status = DIV_OK;
    res->resultado_u.cociente=(p1->dividendo/p1->divisor);
  }
  return res;
}
```

La clase Java que implementa el servicio:

```
// matematicoImpl.java
 import org.omg.PortableServer.*;
 public class MatematicoImpl extends MatematicoPOA {
   public int dividir (int dividendo,int divisor)
         throws MatematicoPackage.DivisionPorCero {
     if (divisor == 0)
       {throw new MatematicoPackage.DivisionPorCero();}
     else
       {return (dividendo/divisor); }
  }
}
```

En esta transparencia se muestra el código, en lenguaje C, que implementa el procedimiento de división para el entorno RPC, así como la clase Java que contiene el procedimiento del servicio matemático que utilizaremos para el entorno de CORBA.

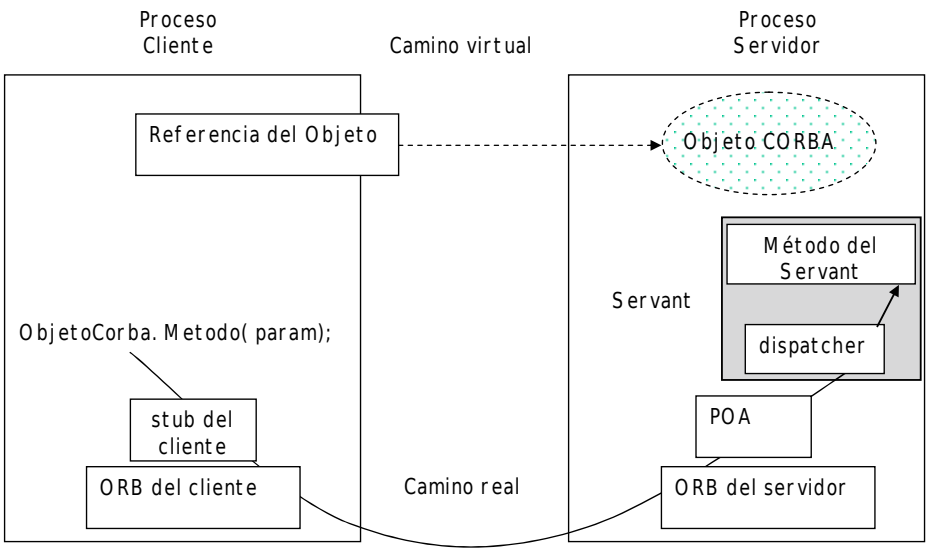
La clase `MatematicoPOA` que se hereda en la clase `MatematicoImpl` se genera automáticamente en la compilación de la interfaz del servicio `Matematico`.

Ahora que tenemos todos los componentes que intervienen en la llamada a un objeto CORBA, veamos en la transparencia siguiente un esquema grafico de una llamada completa.

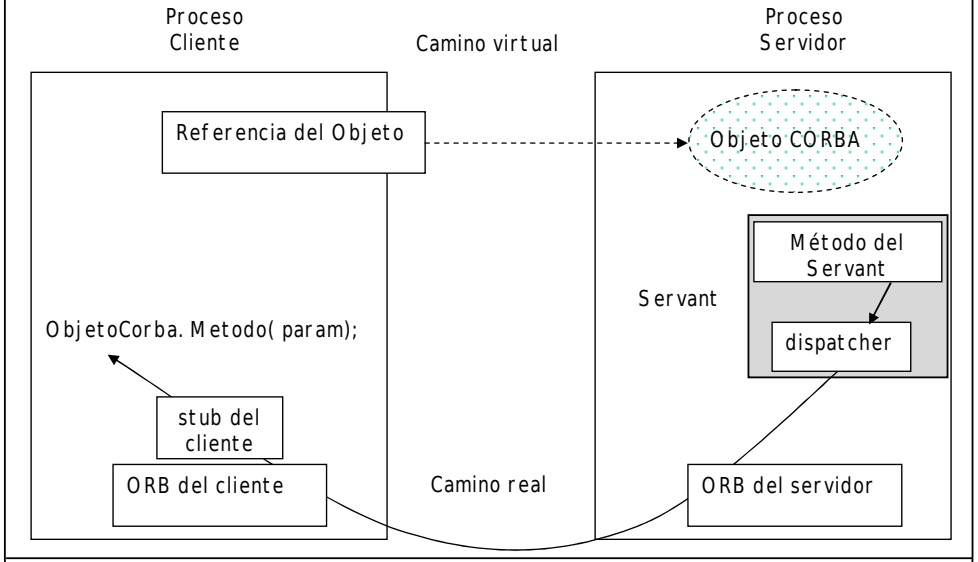
Sistemas Distribuidos La Comunicación - 65

Un Ejemplo en RPC y CORBA...Programa del Servidor

Llamada a un método CORBA:



Envío de la respuesta:



Cuando le llega una petición al servidor, el *dispatcher* es el que se ocupa de llamar al procedimiento adecuado.

Cada objeto tiene su propio *dispatcher*, pues el reparto dentro del *dispacher* tendrá diferentes opciones en nombre y número, dependiendo de los nombres y número de métodos que tenga un objeto.

Pero **¿quién entrega las peticiones al *dispatcher*?**

En el entorno RPC, las peticiones se las entrega directamente el sistema operativo a través de un *socket*.

En CORBA, por cada servicio hay un POA, y de cada servicio puede haber varios objetos creados. Pues bien, cuando llega una petición para uno de los objetos de un servicio, es el POA de ese servicio el que sabe a cual de los objetos debe dirigirlo.

Y **¿quién le entrega el mensaje al POA?**

El bus ORB. Este bus entrega al POA una petición sobre un objeto CORBA con los datos en el formato local de esa máquina. A su vez, el bus ORB ha obtenido el mensaje del modulo de comunicación que haya por debajo, por ejemplo del sistema operativo local.

6. Envío de la respuesta

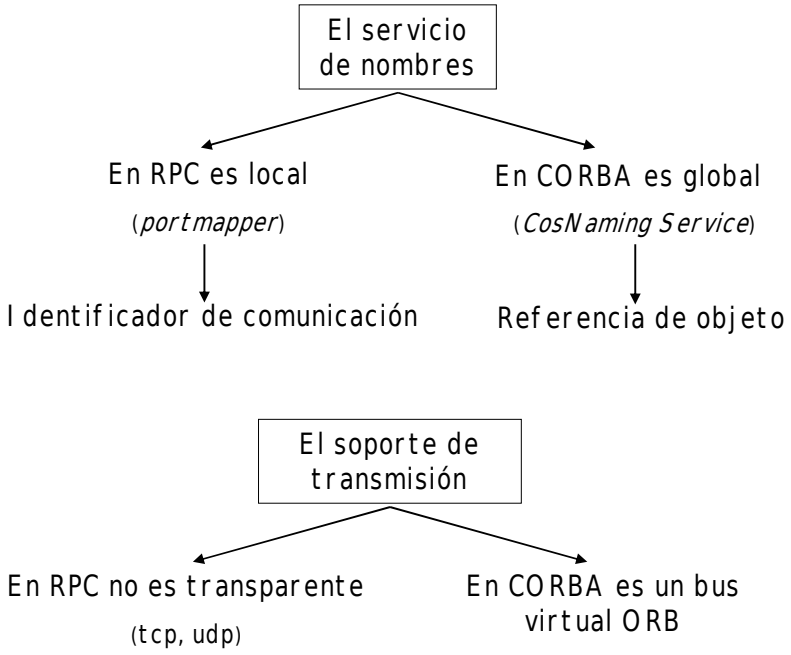
Una vez que el procedimiento adecuado (o rutina de servicio) ha terminado, le devuelve al *dispatcher* el resultado. A continuación, en el caso de RPC, el *dispatcher* serializa el resultado, lo convierte a un formato externo de representación y, por último, lo envía a través de un *socket*. En el caso de CORBA, el resultado en formato local se le pasa al ORB, el cual se encarga de serializarlo y enviarlo al cliente.

Resumiendo

Resumiendo

- En RPC un servidor soporta un servicio
- En CORBA un servicio es un objeto CORBA

Un servidor soporta un objeto CORBA



- La llamada a una operación en RPC es "procedural"
- En CORBA se sigue el modelo `objeto.metodo(parámetros)` presentando transparencia de acceso y de ubicación

Resumiendo, en CORBA un cliente se comunica con un servidor que soporta un objeto CORBA que ofrece un determinado servicio.

Para poder comunicarse con el servidor, el cliente necesita un identificador de comunicación o referencia del objeto CORBA que soporta el servicio. Este identificador lo obtiene del servicio de nombres de CORBA, que es global, por lo que no puede haber dos objetos CORBA registrados con el mismo identificador.

El soporte de transmisión empleado es el bus virtual ORB, es decir, no es un componente hardware para la transmisión de mensajes, es un bus virtual que para conseguir la transmisión física puede apoyarse en las llamadas al sistema que ofrezca el módulo de comunicaciones del sistema operativo donde se ejecute o, incluso, el ORB puede apoyarse en la capa RPC que tenga esa máquina. Este bus virtual, por defecto, presenta la semántica *al menos una* y esconde todos los detalles de comunicación.

Se debe tener cuidado en la implementación del ORB, ya que si se realiza apoyándose en las RPC, la velocidad de transmisión se vería muy afectada, debido al número de capas de software que habría que atravesar hasta llegar al medio físico de transmisión y viceversa.

Por último, diremos que la forma de llamar a los métodos presenta transparencia de acceso y de ubicación ya que independientemente de que el método pertenezca a un objeto local o remoto se invoca de la misma manera:

```
Objeto.método(parámetros)
```