Prácticas de Sistemas Informáticos Distribuidos

Titulación: 3º Ingenierías Técnicas de Sistemas y Gestión

Profesor: José Luis Garrido Bullejos

Dpto. Lenguajes y Sistema Informáticos - Universidad de Granada

Servicios de red y programación distribuida en Unix

1 Servicios de red

1.1 Introducción

Se pretenden estudiar algunas facilidades estándares que proporciona Unix para soportar operaciones distribuidas sobre una red. Todas estas facilidades se basan en el modelo cliente-servidor para aplicaciones distribuidas.

Para localizar un servicio en la red, lo primero que tiene que conocer el cliente es la máquina a contactar. Esto se puede proporcionar bien por una cadena de caracteres o por la dirección de red traducida a un número de 32 bits (dirección IP).

Dentro de cada máquina habrá varios terminales de comunicaciones donde los servidores estarán esperando conexiones. Estos terminales se identifican por un número de **puerto** (normalmente 16 bits). Por ejemplo, el servidor *sendmail* siempre espera conexión en el puerto 25, y el demonio *rexecd* que ofrece un servicio de ejecución de orden remota en el puerto 512. La correspondencia entre servicios y puertos está almacenada en el archivo */etc/services*. Aquí también se incluye el protocolo que se utiliza y un nombre alternativo para el servicio.

Este método de localizar un servicio es extremadamente primitivo, ya que no se puede proporcionar un nombre de servicio y pedir al sistema que localice una máquina que ofrezca ese servicio (esto sí existe para sistemas de archivos compartidos como RFS de AT&T, y también en el entorno DCE de OSF el cual proporciona un servicio directorio de celda).

El número de puerto, por ejemplo de *sendmail*, se dice que es **conocido** debido a que este servidor, en todas las máquinas, siempre espera conexión en ese puerto.

Cuando un cliente se conecta a un servidor tiene que crear otro terminal de transporte de su propiedad, el cual tendrá también asignado un número de puerto que será arbitrario, con la excepción de que los números menores de 1024 son **reservados** (sólo procesos ejecutándose con *uid* efectivo de *root* podrán utilizar estos números de puertos). La mayoría de los servidores utilizan puertos reservados. Ésta es la base para proporcionar seguridad a bajo nivel en los servicios de red. Algunos servidores piden que los clientes utilicen puertos reservados cuando intentan la conexión, lo cual asegura un tipo cliente concreto.

1.2 Ejecución de orden remota

La ejecución de órdenes remotas es uno de los principales servicios de red que ofrecen los sistemas operativos. A continuación se dan las diferentes posibilidades para ejecución de órdenes remotas:

1) La principal orden de ejecución remota es *rsh*, lógicamente éste es el programa cliente que se ejecuta en la máquina local. El servidor se denomina *rshd*. El formato de la orden es:

```
rsh <máquina remota> <orden>
```

   El método que utiliza el servidor para comprobar la identidad del usuario se denomina **máquinas de confianza**. Este método evita tener que especificar el nombre de usuario y la clave cada vez que se ejecuta una orden remota. Así, para poder ejecutar la orden remota se tienen que satisfacer dos condiciones:

   a. El usuario tiene que tener una cuenta en la máquina remota (normalmente con el mismo nombre de usuario).

   b. El archivo */etc/hosts.equiv* de la máquina remota tiene que tener una entrada con el nombre de la máquina local, o si esto falla, en el directorio de la cuenta en la máquina remota tiene que haber un archivo *.rhosts* que contenga dicha entrada.

2) Una variante de la orden *rsh* es *on*. Es un servicio basado en RPC soportado por del demonio *rexd* y tiene el mismo formato que *rsh*. La diferencia es que crea un entorno en la máquina remota similar al que hay en la máquina local, por ejemplo, todas las variables de entorno locales se pasan explícitamente a la máquina remota. Otra diferencia es que soporta órdenes remotas interactivas, por ejemplo *vi*.

3) Utilizar el servidor *rexecd*. Su ejecución no depende de que el cliente tenga privilegios de supervisor. Este servidor no tiene un programa cliente, pero tiene una función *rexec()* para que los usuarios puedan escribir sus propios programas cliente. Aquí la identificación se realiza introduciendo explícitamente el nombre de usuario y la clave cada vez que se llama al servidor.

4) Utilizar el cliente de propósito general *telnet*. Normalmente, *telnet* conecta con el servidor *telnetd* en el número de puerto TCP 23, pero opcionalmente, se puede utilizar con un número de puerto alternativo donde se encuentre algún otro servicio orientado a texto (para obtener salidas legibles). Por ejemplo, la ejecución de orden remota siguiente devuelve la fecha y hora de la máquina remota *ws1*:

```
telnet ws1 daytime
```

   Se pueden obtener los servicios que ofrece actualmente una máquina con la orden:

```
netstat -a
```

La siguiente tabla resume los métodos de ejecución remota:

Servidor	Función cliente	Aplicación cliente	Comentarios
rshd	rcmd()	rsh, rcp	Parte de las utilidades r* de BSD. Identificación basada en el uso de puertos reservados y en el archivo /etc/hosts.equiv
rexd	rex()	on	Servicio basado en RPC. Pasa datos del entorno y soporta uso interactivo
rexecd	rexec()	No existen	Similar a rshd, pero la identificación por explicito nombre de usuario y clave
El que se especifique	No se conoce	telnet	El servicio solicitado debe de ser orientado a texto para obtener una salida legible

1.3 Ejemplo

Pruebe el siguiente cliente que conecta con el servidor *rexecd* para llevar a cabo la ejecución remota de cualquier orden que se pase como parámetro a dicho cliente. Consulte el formato de la función *rexec* con la orden *man*. Además, observe como se utilizan las funciones *read* y *fwrite* para obtener la información de la red e imprimirla en pantalla respectivamente.

```
#include <sys/types.h>
#include <stdio.h>
#include <netinet/in.h>

main (int argc, char *argv[]) {

int fd, count;
char buffer[BUFSIZ];

fd = rexec(&argv[1], 512, 0, 0, argv[2],0);
if (fd == -1) {
       printf("Ha fallado rexec\n");
       exit(1);
}
while ((count=read(fd, buffer, BUFSIZ))>0)
       fwrite(buffer, count, 1, stdout);
}
```

2 Llamada remota a procedimiento (RPC)

2.1 Introducción

La independencia del transporte de RPC aísla a las aplicaciones de los elementos físicos y lógicos de los mecanismos de comunicaciones de datos y permite a la aplicación utilizar varios transportes. RPC permite a las aplicaciones de red utilizar llamadas a procedimientos que ocultan los detalles de los mecanismos de red subyacentes.

Características de RPC de Sun:

1) Cada procedimiento RPC se identifica unívocamente. Se lleva a cabo mediante:
   a. Un **número de programa** que identifica un grupo de procedimientos remotos relacionados.
   b. Un **número de versión** para que cuando se realicen cambios en un servicio remoto (p.e. añadir un nuevo procedimiento) no tenga que asignarse un nuevo número de programa.
   c. Un **número de procedimiento** único.
2) Selección de red. Permite a los usuarios y aplicaciones seleccionar dinámicamente un transporte de los disponibles. Para esto se utilizan dos mecanismos:
   a. El archivo */etc/netconfig* que contiene una lista de transportes disponibles para la máquina y los identifica por tipos:
      - Campo 1: es el identificador de red del transporte.
      - Campo 2: tipo de transporte, *tpi_clts* son sin conexión, *tpi_cots* con conexión y *tpi_cots_ord* conexión con ordenación.
      - Campo 3: contiene *flags* que identifican el tipo de transporte (p.e. *v* para uno que pueda ser seleccionado).
      - Último campo: contiene uno más módulos enlazables de tiempo de ejecución que contienen las rutinas de traducción de nombres a direcciones.
      - Los transportes *loopback* se utilizan para registrar servicios con *rpcbind* y son sólo transportes locales.
   b. La variable de entorno *NETPATH* especifica el orden en el cual la aplicación intenta los transportes disponibles. Su formato es una lista ordenada de identificadores de red

separados por dos puntos (p.e. *tcp:udp*). RPC selecciona el transporte según se especifique:

- *netpath*. Escoge los transportes especificados en la variable de entorno del mismo nombre. Si ésta no se ha definido, utiliza los transportes del archivo */etc/netconfig* que tengan el *flag v* y según el orden en que aparecen.
- *visible*. Similar al caso anterior cuando utiliza el archivo.
- *tcp*. Utiliza el protocolo TCP/IP.
- *udp*. Utiliza el protocolo UDP.

3) Utilidad *rpcbind* que en versiones anteriores se denominaba *portmap*. Los servicios de transporte no proporcionan servicios de búsqueda de direcciones, sólo proporcionan transferencia de mensajes a través de la red. Esta utilidad proporciona el medio para que un cliente pueda obtener la dirección de un programa servidor. Proporciona las siguientes operaciones:
   - Registro de direcciones. Asocia servicios RPC con direcciones. *rpcbind* es el único servicio RPC que tiene dirección conocida (puerto número 111 tanto para TCP como UDP). Un servicio hace su dirección disponible a los clientes cuando la registra con *rpcbind*. Ningún cliente ni servidor puede asumir las direcciones de red de un servicio RPC. La biblioteca RPC (*libnsl*) proporciona una interfaz a todos los procedimientos de *rpcbind*.
   - Borrar registros.
   - Obtener la dirección de un especifico programa.
   - Obtener la lista de registros completa. La lista de registros se puede obtener con la orden *rpcinfo*.
   - Realizar una llamada remota para un cliente.
   - Devolver la hora.
4) Varios niveles. Los servicios de RPC se pueden utilizar a diferentes niveles. Los servicios del nivel más bajo no son necesarios cuando se utiliza la utilidad *rpcgen* para generar los programas RPC.
5) Representación externa de datos (XDR). Para que RPC funcione en varias arquitecturas de sistemas requiere una representación de datos estándar. XDR es una descripción de datos independiente de la máquina y un protocolo de codificación. Con XDR, RPC consigue manejar estructuras de datos arbitrarias sin importar la ordenación de *bytes* que haga cada máquina o las cuestiones en el diseño de estructuras.

2.2 rpcgen

Esta utilidad genera módulos interfaz de programas remotos compilando código fuente escrito en el lenguaje RPC. Por defecto, la salida de *rpcgen* es:

- Un archivo cabecera con las definiciones comunes al servidor y al cliente.
- Un conjunto de rutinas XDR que traducen cada tipo de dato definido en el archivo cabecera.
- Un programa *stub* para el servidor y otro para el cliente.

Además, *rpcgen* opcionalmente genera una tabla informe de RPC para comprobar autorizaciones y para invocar rutinas de servicio. Otras opciones incluyen especificar plazos de tiempo para servidores, seleccionar varios transportes, código conforme al ANSI-C pasando argumentos estilo C.

Pasos en el desarrollo de un programa distribuido:

1) Determinar los tipos de datos de todos los argumentos del procedimiento llamado y escribir la especificación del protocolo en el lenguaje RPC. Por ejemplo, para un procedimiento llamado *printmessage()* al cual se le pasa una cadena de caracteres y devuelve un entero, el código fuente para la especificación del protocolo sería:

```
/* Archivo msg.x: Protocolo de impresion de un mensaje remoto */
   program MESSAGEPROG {
     version PRINTMESSAGEVERS {
       int PRINTMESSAGE (string) = 1;
     } = 1;
   } = 0x20000001;
```

Este protocolo declara el procedimiento *PRINTMESSAGE* como procedimiento número 1, en la versión número 1 del programa *MESSAGREPROG*, cuyo número de programa es *0x20000001*. Los números de programa se dan según la siguiente tabla:

0 - 1fffffff	Definidos por Sun
20000000 - 3fffffff	Definidos por los usuarios para programas particulares
40000000 - 5fffffff	Reservados para programas que generan números de programas dinámicamente
60000000 - ffffffff	Reservados para uso futuro

Por convención, los nombres se suelen escribir en mayúsculas y el nombre del archivo que contiene la especificación acaba en *.x*.

El tipo de argumento *string* se corresponde con *char* * de C.

2) Escribir el procedimiento remoto y el programa cliente principal que lo llama.

```
/* msg_proc.c: implementacion del procedimiento remoto */

  #include <stdio.h>
  #include "msg.h"              /* el archivo lo genera rpcgen */

  int *
  printmessage_1(msg, req)
    char **msg;
    struct svc_req *req;  /* detalles de la llamada */

    {
      static int result;   /* es obligatorio que sea estatica */
      FILE *f;
      f = fopen("/dev/console", "w");
      if (f == (FILE *)NULL) {
        result = 0;
        return (&result);
      }
      fprintf(f,"%s\n", *msg);
      fclose(f);
      result = 1;
      return (&result);
    }
```

Observe lo siguiente en la declaración del procedimiento remoto:

- Toma un puntero a un array de caracteres. Esto ocurre cuando no se utiliza la opción *-N* con *rpcgen*. Sin esta opción, los procedimientos remotos siempre se llaman con un único argumento, si es necesario más de uno se pasan en una estructura.
- El segundo parámetro contiene información sobre el contexto de una invocación (programa, versión, procedimiento, un puntero a una estructura que contiene información de transporte, ...). Esta información se hace disponible para el caso de que el procedimiento invocado la requiera para realizar la petición.
- Devuelve un puntero a un entero en lugar del entero. Esto también ocurre cuando no se utiliza la opción *-N* con *rpcgen*. Siempre se declara el resultado como *static*, ya que si se

declara local al procedimiento remoto, las referencias a él por el *stub* del servidor son inválidas después de la vuelta del procedimiento.

- Un *_1* se añade al nombre según el número de versión, lo cual permite múltiples versiones del mismo nombre.

```
/* Archivo  rprintmsg.c: programa cliente */
  #include <stdio.h>
  #include "msg.h"

  main(argc, argv)
    int argc;
    char *argv[];
  {
    CLIENT *clnt;
    int *result;
    char *server;
    char *message;
    if (argc != 3) {
      fprintf(stderr, "uso: %s maquina mensaje\n", argv[0]);
      exit(1);
    }
    server = argv[1];
    message = argv[2];

    /* Crea estructura de datos(handle) del proceso
       cliente para el servidor designado     */
    clnt = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,"visible");
    if (clnt == (CLIENT *)NULL) {
      /* No se pudo establecer conexion con el servidor */
      clnt_pcreateerror(server);
      exit(1);
    }

    /* Llamada remota al procedimiento en el servidor */
    result = printmessage_1(&message, clnt);
    if (result == (int *)NULL) {
      /* Ocurrio un error durante la llamada al servidor */
      clnt_perror(clnt, server);
      exit(1);
    }
    if (*result == 0) {
      /* El servidor fue incapaz de imprimir nuestro mensaje */
      fprintf(stderr, "%s: no pudo imprimir el mensaje\n", argv[0]);
      exit(1);
```

```
    }
    printf("Mensaje enviado a %s\n", server);
    clnt_destroy( clnt );
    exit(0);
}
```

Observe lo siguiente en la declaración del cliente:

- La rutina de biblioteca *clnt_create()* crea una estructura de datos (*handle*) del cliente. Ésta se pasa a la rutina *stub* que llama al procedimiento remoto. Cuando no se van a realizar más llamadas se destruye la estructura de datos con *clnt_destroy()* para conservar los recursos del sistema.
- El último parámetro *visible* de *clnt_create()* especifica cualquier transporte con el *flag v* en el archivo */etc/netconfig*.

3) Generación del archivo de cabecera *msg.h*, *stub* del cliente (*msg_clnt.c*) y *stub* del servidor (*msg_svc.c*) con:

```
rpcgen msg.x
```

4) Compilación de los dos programas, cliente y servidor, y enlazado de cada uno con la biblioteca *libnsl* que contiene todas las funciones de red (incluyendo RPC y XDR):

```
cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
cc msg_proc.c msg_svc -o msg_server -lnsl
```

5) Ejecución del servidor y posteriormente del cliente. El servidor generado por rpcgen siempre se ejecutará en segundo plano (background) sin necesidad de invocarlo con &. Además estos servidores pueden ser invocados por monitores de puertos como *inetd*.

2.3 Ejemplo completo de servicio RPC

En el ejemplo de la sección anterior no fueron generadas rutinas XDR (archivo *msg_xdr.c*) por *rpcgen* debido a que el programa sólo utiliza tipos básicos que están incluidos en *libnsl*. Si se hubieran definido tipos de datos en el archivo *.x* si se habrían generado estas rutinas. Estas rutinas se utilizan para convertir estructuras de datos locales al formato XDR y viceversa. Por cada tipo de dato definido en el archivo *.x*, *rpcgen* genera una rutina con el prefijo *xdr_* que se incluye en el archivo *_xdr.c*. A continuación se presenta un servicio RPC de listado de directorio remoto:

```
/* dir.x : Protocolo de listado de directorio remoto */
  const MAX= 255;       /* longitud maxima de la entrada directorio */
  typedef string nametype<MAX>;               /* entrada directorio */
  typedef struct namenode *namelist;        /* enlace en el listado */

  struct namenode{
    nametype name;        /* nombre de la entrada de directorio */
    namelist next ;                         /* siguiente entrada */
  };

  /* La siguiente union se utiliza para discriminar entre llamadas
   *                 con exito y llamadas con errores              */
  union readdir_res switch (int errno) {
    case 0:
      namelist list; /* sin error: listado del directorio */
    default:
      void;          /* con error: nada                   */
  };

  program DIRPROG {
    version DIRVER {
      readdir_res READDIR(nametype) = 1;
    } =1;
  } = 0x20000155;
```

Se pueden redefinir tipos (como *readdir_res*) usando las palabras reservadas *struct*, *union* y *enum* (en una sección posterior se introduce el lenguaje RPC). *rpcgen* compila uniones RPC en estructuras de C. Utilizando *rpcgen* sobre *dir.x* genera, además de los archivos ya vistos, el archivo *dir_xdr.c*.

Para generar los *stubs*, plantillas, ..., ejecutaremos (todas las opciones de *rpcgen* se comentan en la siguiente sección):

```
turing% rpcgen -NCa dir.x
```

```
/* dir_client.c: codigo del cliente       */
  /*
   * This is sample code generated by rpcgen.
```

```
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "dir.h"             /* creado por rpcgen */
extern int errno;

void
dirprog_1(char *server,  nametype dir)
{
  CLIENT *clnt;
  readdir_res  *result;
  namelist nl;

  #ifndef DEBUG
  clnt = clnt_create(server, DIRPROG, DIRVER, "netpath");
  if (clnt == (CLIENT *) NULL) {
    clnt_pcreateerror(server);
    exit(1);
  }
  #endif  /* DEBUG */

  result = readdir_1(dir, clnt);
  if (result == (readdir_res *) NULL) {
    clnt_perror(clnt, server);
    exit(1);
  }

  if (result->errno != 0) {
    errno =result->errno;
    perror(dir);
    exit(1);
  }

  for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->next)
    printf("%s\n", nl->name);

  xdr_free (xdr_readdir_res,  result);

  #ifndef DEBUG
  clnt_destroy(clnt);
  #endif          /* DEBUG */

  exit(0);
}
 main(int argc, char *argv[])
```

```
{
  char *server;
  nametype dir;
  setbuf(stdout, NULL);
  if (argc < 3) {
    printf("usage:  %s server_host directory\n", argv[0]);
    exit(1);
  }
  server = argv[1];
  dir = argv[2];
  dirprog_1(server, dir);
}
```

Observe que el código de cliente generado por *rpcgen* no libera la memoria asignada por la llamada RPC, para esto se utiliza *xdr_free()*. Es similar a la función *free()*, excepto en que se pasa la función XDR para el resultado devuelto.

```
/* dir_server.c : codigo del servidor   */
  /*
   * This is sample code generated by rpcgen.
   * These are only templates and you can use them
   * as a guideline for developing your own functions.
   */
  #include "dir.h"              /* creado por rpcgen */
  #include <dirent.h>

  extern int errno;

  readdir_res *
  readdir_1_svc(nametype dirname, struct svc_req *rqstp)
  {
    DIR *dirp;
    struct dirent *d;
    namelist nl;
    namelist *nlp;

    static readdir_res  result;     /* tiene que ser estatica */

    dirp = opendir(dirname);
    if  (dirp == (DIR *)NULL) {
      result.errno = errno;
      return (&result);
```

```
    }
    /*
     * La siguiente linea de codigo libera la memoria que se asigno
     * (para el resultado) en una ejecucion previa del servidor
     */
    xdr_free(xdr_readdir_res, &result);

    nlp = &result.readdir_res_u.list;
    while (d = readdir(dirp)) {
      nl = *nlp = (namenode *) malloc(sizeof(namenode));
      if (nl == (namenode *) NULL) {
        result.errno =  10;
        closedir(dirp);
        return (&result);
      }
      nl->name = strdup(d->d_name);
      nlp = &nl->next;
    }
    *nlp = (namelist)NULL;
    result.errno = 0;
    closedir (dirp);
    return (&result);
}
```

Los archivos serán compilados como sigue:

```
turing% cc dir_client.c dir_clnt.c dir_xdr.c -o cliente -lnsl
turing% cc dir_server.c dir_svc.c dir_xdr.c -o servidor -lnsl
```

o como alternativa a lo anterior:

```
turing% make -f makefile.dir
```

Y el programa se ejecutará como sigue:

```
ws1% servidor
turing% cliente ws1 /etc
```

Este ejemplo, también ilustra como resuelve *rpcgen* por si sólo el problema de que pasar punteros del espacio de direcciones de una máquina a otra, en la cual no serían válidos. XDR proporciona en el archivo *_xdr.c* una función filtro especial denominada *xdr_pointer()*, el cual es capaz de seguir cadenas de punteros y codificar el resultado en una cadena de *bytes*. Esto funciona correctamente en algunos tipos recursivos de estructuras de datos tales como listas enlazas y árboles binarios, ya que proporcionan un final mediante punteros nulos, con lo cual son estructuras enlazadas acíclicas. Sin embargo, esta técnica no funciona con estructuras de datos cíclicas tales como listas circulares, listas doblemente enlazadas, ...

2.4 Opciones de rpcgen

Las opciones de tiempo de compilación son:

- Código estilo C (*-N*). Provoca que *rpcgen* genere código con los argumentos pasados por valor y si hay varios argumentos estos se pasan sin *struct*.

```
/* archivo add.x: Demuestra el paso de argumentos por defecto.
                  Solo se puede pasar un argumento    */
  struct add_arg {
    int first;
    int second;
  }

  program ADDPROG {
    version ADDVER {
      int ADD (add_arg) = 1;
    } = 1;
  } = 0x20000199;
/* archivo add.x: Demuestra el paso de argumento estilo C */

  program ADDPROG {
    version ADDVER {
      int ADD (int, int) = 1;
    } = 1;
  } = 0x20000199;
```

- Archivos plantilla. Genera código ejemplo que puede servir como guía o se puede utilizar directamente rellenando las partes omitidas. Posibilidades:

Opción	Función
`-a`	Genera todos los archivos plantilla
`-Sc`	Genera la plantilla para el cliente
`-Ss`	Genera la plantilla para el servidor
`-Sm`	Genera la plantilla del archivo para la utilidad make

- Código compatible con ANSI-C o SPARCompiler C++ 3.0 (*-C*). Esta opción se suele utilizar junto con la opción *-N*. Observe que los nombres de los procedimientos remotos en el servidor requieren el sufijo *_svc*.

- Código MT-seguro (*-M*). Por defecto el código generado por *rpcgen* no es MT-seguro, ya que utiliza variables globales sin proteger y devuelve resultados mediante variables estáticas. Con esta opción se genera código MT-seguro el cual puede utilizarse en un entorno multi-hebras. La opción se puede utilizar junto con las opciones *-N* y *-C*.

2.5 Referencia del lenguaje RPC

Este lenguaje es una extensión del lenguaje XDR. A continuación se describe su sintaxis junto con algunos ejemplos y el resultado de la compilación de las definiciones de tipos del lenguaje RPC, lo cual se incluye en el archivo de cabecera *.h* de la salida de *rpcgen*.

El lenguaje consiste en una serie de definiciones:

```
<lista-definiciones> ::= <definicion>; | <lista-definiciones>;
<definicion> ::= <definicion-enum> |
                 <definicion-const> |
                 <definicion-typedef> |
                 <definicion-struct> |
                 <definicion-union> |
                 <definicion-program>
```

Definiciones no son lo mismo que declaraciones, no se asigna espacio en una definición (sólo la definición de tipo de uno o varios elementos de datos). Es decir, las variables tendrán que ser declaradas posteriormente en el programa.

2.5.1 Enumeraciones

Tienen la misma sintaxis que las de C:

```
<definicion-enum> ::= enum <ident-enumeracion> "{"
                            <lista-valores-enum>
                      "}"
<lista-valores-enum> ::= <valor-enum> |
                         <valor-enum> , <lista-valores-enum>
<valor-enum> ::= <ident-valor-enum> |
                 <ident-valor-enum> = <valor>
```

A continuación se da un ejemplo de definición y su traducción a C:

```
enum tipocolor {                        enum tipocolor {
   ROJO = 0,                               ROJO = 0,
   VERDE = 1,            --->              VERDE = 1,
   AZUL = 2                                AZUL = 2,
};                                      };
                                        typedef enum tipocolor tipocolor;
```

2.5.2 Constantes

Las constantes simbólicas pueden usarse donde una constante entera se usa.

```
<definicion-const> ::= const <ident-constante> = <entero>
```

Ejemplo:

```
const DOCENA = 12;          --->      #define DOCENA 12
```

2.5.3 Definiciones de Tipos

Tienen exactamente la misma sintaxis que en C:

```
<definicion-typedef> ::= typedef <declaracion>
```

El ejemplo siguiente define un tipo utilizado para declarar cadenas de nombres de archivo que tienen una longitud máxima de 255 caracteres:

```
typedef string tipo_nombref<255>;   --->   typedef char *tipo_nombref;
```

2.5.4 Declaraciones

No se debe confundir declaraciones de variables con declaraciones de tipos. *rpcgen* no soporta declaraciones de variables. Hay cuatro clases de declaraciones. Estas declaraciones tienen que ser parte de un *struct* o un *typedef*; no se pueden encontrar solas:

```
<declaracion> ::= <declaracion-simple> |
                  <declaracion-array-fijo> |
                  <declaracion-array-variable> |
                  <declaracion-puntero>
<declaracion-simple> ::= <ident-tipo> <ident-variable>
```

Ejemplo de declaración simple:

```
tipocolor color;          --->        tipocolor color;
```

```
<declaracion-array-fijo> ::= <ident-tipo>  <ident-variable> "["<valor>"]"
```

Ejemplo de array de tamaño fijo:

```
tipocolor paleta[8];      --->        tipocolor paleta[8];
```

La declaración de arrays variables no tiene sintaxis explicita en C. Se puede especificar un tamaño máximo entre los ángulos. Estas declaraciones se traducen a declaraciones *struct* de C.

```
<declaracion-array-variable>::=<ident-tipo> <ident-variable> "<"valor">" |
                               <ident-tipo> <ident-variable> "<" ">"
```

Ejemplo:

```
int altura<12>;          --->       struct {
                                      u_int altura_len;
                                       int *altura;
                                    } altura;
```

La declaración de punteros es igual a la de C. Los punteros de direcciones no se envían realmente por la red, pero son útiles para enviar tipos de datos recursivos tales como listas o árboles.

```
<declaracion-puntero> ::= <ident-tipo> *<ident-variable>
```

Ejemplo:

```
listaelementos *siguiente;     --->    listaelementos *siguiente;
```

2.5.5 Estructuras

Se declaran igual que en C.

```
<defincion-struct> ::= struct <ident-estructura> "{"
                                <lista-declaraciones>
                          "}"
<lista-declaraciones> ::= <declaracion> ; |
                            <declaracion> ; <lista-declaraciones>
```

Ejemplo:

```
struct coord {                        struct coord
  int x;       --->                     int x;
```

```
  int y;                                    int y;
};                                        };
                                          typedef struct coord coord;
```

2.5.6 Uniones

Las uniones de RPC son diferentes a las uniones de C. Son similares a los registros variantes de PASCAL.

```
<definicion-union>::=union <ident-union> switch (<declaracion-simple>) "{"
                          <lista-case>
                     "}"
<lista-case> ::= case <valor> : <declaracion> ; |
                 case <valor> : <declaracion> ; <lista-case> |
                 default : <declaracion> ;
```

El siguiente es un ejemplo de un tipo devuelto como resultado de una operación de lectura de un dato: si no hay error, devuelve un bloque de datos; en otro caso, no devuelve nada.

```
union resultado_leido switch (int errno) {
      case 0:
        tipodato dato;
      default:
        void;
};
```

Y se traduce a C:

```
struct resultado_leido {
      int errno;
      union {
        tipodato dato;
      } resultado_leido_u;
};
typedef struct resultado_leido resultado_leido;
```

2.5.7 Programas

Los programas RPC se declaran con la siguiente sintaxis:

```
<definicion-program> ::= program <ident-programa> "{"
                                  <lista-versiones>
                            "}" = <valor>;
<lista-versiones> ::= <version> ; |
                      <version> ; <lista-versiones>
<version> ::= version <ident-version> "{"
                      <lista-procedimientos>
              "}" = <valor> ;
<lista-procedimientos> ::= <procedimiento> ; |
                           <procedimiento> ; <lista-procedimientos>
<procedimiento> ::= <ident-tipo>  ( <ident-tipo> ) = <valor>;
```

Cuando se especifica la opción *-N* en *rpcgen*, reconoce la siguiente sintaxis:

```
<procedimiento> ::= <ident-tipo>  ( <lista-ident-tipo> ) = <valor>;
<lista-ident-tipo> ::= <ident-tipo> |
                       <ident-tipo> , <lista-ident-tipo>
```

2.5.8 Casos especiales

- **Lógicos**. La biblioteca RPC utiliza un tipo lógico llamado *bool_t* que es *TRUE* o *FALSE*. Los parámetros declarados de tipo *bool* en el lenguaje RPC se traducen a *bool_t* en el archivo de cabecera de salida de *rpcgen*.

- **Cadenas de caracteres**. En el lenguaje RPC las cadenas de caracteres se declaran con la palabra reservada *string*, y se traduce al tipo *char* * en el archivo de cabecera. El tamaño máximo contenido entre los ángulos especifica el número máximo de caracteres permitidos en la cadena de caracteres (sin contar el carácter *NULL*). El tamaño máximo puede quedar sin implementación, indicando una cadena de caracteres de longitud arbitraria. Ejemplo:

```
string nombre<32>;          --->     char *nombre;
string nombrecompleto<>;    --->     char *nombrecompleto;
```

Observe que cadenas de caracteres *NULL* no pueden pasarse; sin embargo, una cadena de longitud cero sí (sólo con el byte *NULL*).

- **Datos opacos**. Se utilizan para describir datos sin tipo, es decir, secuencias de *bytes* arbitrarios. Pueden declararse como un array bien de longitud fija o variable. Ejemplos:

```
opaque bloquedisco[512];  --->   char bloquedisco[512];
opaque datosf<1024>;      --->   struct {
                                     u_int datosf_len;
                                     char *datosf_val;
                                 } datosf;
```

- ***void***. Declaraciones *void* pueden tener lugar sólo en dos lugares:

  – definiciones de uniones

  – definiciones de programas, como argumento (p.e. indicando que no se pasan argumentos) o resultado de un procedimiento remoto.

2.6 Ejercicio 1

Escriba un programa distribuido, utilizando *rpcgen*, que realice las operaciones aritméticas suma, resta, multiplicación o división sobre enteros, según las siguientes especificaciones:

- El cliente se encargará de filtrar la línea de orden introducida por el usuario la cual podrá tener el siguiente formato:

```
<programa> <maquina> <entero> <operador> <entero>
```

  donde <*operador*> puede ser + | - | *x* | /.

  El programa cliente será el encargado de llamar a la operación correspondiente en el servidor.

- El servidor sólo realizará la evaluación de la expresión y devolverá el resultado.

2.7 Ejercicio 2

Consiste en programar un **servicio básico de asociación**, en el cual las asociaciones se almacenarán en listas enlazadas.

2.7.1 Conjuntos de Asociaciones

El programa mantiene conjuntos de asociaciones. Una asociación es simplemente una correspondencia de una **clave** a un **valor**. Cada asociación incluye además un **identificador** (id), lo cual significa que se pueden almacenar diferentes conjuntos de asociaciones (p.e. nombres a números de teléfono, palabras españolas a inglesas, ...).

2.7.2 Almacenamiento y recuperación de Asociaciones

Una asociación se define por una tupla de valores *(id, clave, valor)* cuyos tipos se definen en la siguiente tabla:

Nombre	Nombre del tipo	Descripción
id	ID	un entero
clave	Clave	un puntero a un string terminado nulo
valor	Valor	un puntero a un string terminado nulo
s	Estado	Indica el resultado de la operación (p.e., OK, Sustitución, NoEncontrado, ...

El programa se estructurará en dos módulos:

1) Módulo cliente para **interpretar órdenes**. Proporciona al usuario búsqueda y recuperación del valor de una clave, añadir, borrar y reemplazar asociaciones dentro de un conjunto concreto.
2) Módulo servidor para el **manejo de asociaciones**.

2.7.3 Interfaz del servicio

El servidor proporcionará las siguientes funciones:

Nombre	Descripción
PonerAsociacion: ID x Clave x Valor -> Estado	Provoca que la información se almacene para futuras recuperaciones, si ya exite una tupla con la misma clave, el nuevo valor debe de reemplazar al antiguo. El estado devuelto debe de distinguir entre estos dos casos.

ObtenerAsociacion: ID x Clave -> Valor x Estado	Obtiene el valor de una asociación. El valor de estado distinguirá entre éxito o fallo
BorrarAsociacion: ID x Clave -> Estado	Provoca la eliminación de una asociación. Estado indicará si se encontró la asociación
Enumerar: ID -> Conjunto de Claves y Valores x Estado	Devuelve todas las claves asociadas a un conjunto y sus correspondientes valores

Lenguajes y *Middlewares* para Programación Distribuida

3 Java y RMI

3.1 Introducción

RMI (*Remote Method Invocation*) es una de las dos formas que Java propone para trabajar con objetos distribuidos. La otra alternativa es mediante la interfaz Java-IDL. IDL (*Interface Definition Language*) se ha diseñado para comunicar entre objetos Java y objetos creados en otros lenguajes (e.g. C++) mediante *Object Request Brokers* (normalmente aquellos que satisfacen el estándar CORBA).

Suponga que desea coleccionar información localmente en un cliente y enviarla a través de la red a un servidor. Por ejemplo, un usuario rellena un formulario de petición de información, el cual se envía al vendedor, y este devuelve la información del producto solicitado. Para hacer esto, existen varios métodos:

1) **Conexiones con *sockets*** para enviar flujos de bytes entre el cliente y el servidor. Este método es útil si sólo es necesario enviar datos en bruto a través de la red.
2) **JDBC** para realizar consultas y actualizaciones de bases de datos. Es útil cuando la información que se envía encaja en el modelo de tablas de una base de datos relacional.
3) **RMI** se basa en la implementación del formulario de petición y de la información del producto como objetos. Utilizando este método los objetos pueden ser transportados entre el cliente y el servidor.

Utilizando RMI se obtienen los siguientes beneficios:

- El programador no tiene que manejar flujos de bytes.
- Se pueden utilizar objetos de cualquier tipo.
- Se pueden invocar llamadas a métodos en objetos situados en otra computadora sin tener que desplazarlos a la computadora que realiza la invocación.

Al igual que con RPC, la terminología cliente-servidor se aplica sólo a la llamada de un único método. Las funciones de los procesos se pueden invertir, y así el servidor de una llamada previa puede actuar como cliente en la siguiente llamada.

3.2 Invocaciones Remotas de Métodos

3.2.1 Cómo RMI amplia las llamadas locales

Los clientes RMI interaccionan con los objetos remotos por medio de interfaces. Nunca interaccionan directamente con las clases que implementan estas interfaces.

A diferencia de llamadas locales de Java, una invocación remota pasa los objetos locales en los parámetros por copia, en lugar de por referencia. Por otro lado, ya que se puede acceder a objetos remotos en una red, RMI pasa un objeto remoto por referencia, no copiando la implementación remota real.

RMI proporciona nuevas interfaces y clases que permiten encontrar objetos remotos, cargarlos y después ejecutarlos de forma segura. RMI incluye un **servicio de denominación** (ligadura) básico y no permanente que permite localizar objetos remotos. También proporciona un cargador de clases que deja a los clientes descargar *stubs* desde el servidor. El *stub* sirve como apoderado o sustituto (*proxy*) del objeto remoto en el cliente. RMI proporciona mecanismos de seguridad extra para asegurar un buen comportamiento de estos *stubs*.

Ya que las invocaciones remotas de métodos pueden fallar por muchas más razones que las locales, se han de manejar excepciones adicionales. Por ello, RMI amplia las clases de excepciones de Java para tratar con cuestiones remotas.

Un buen recolector de objetos no utilizados tiene que ser capaz de borrar automáticamente objetos remotos que no se referencian por ningún cliente. RMI utiliza un esquema de recolección basado en contadores de referencias que mantiene la pista de todas las referencias externas existentes a objetos remotos. En este caso, una referencia es justo una conexión cliente-servidor sobre una sesión TCP/IP.

3.2.2 Stubs

Cuando un cliente desea invocar un método en un objeto remoto, realmente llama un método normal Java que hay encapsulado en un objeto sustituto llamado *stub*. Java utiliza un mecanismo de serialización de objetos para formar los parámetros en el objeto *stub*. El método que se invoca en el *stub* del cliente construye un bloque de información que consiste en:

- Un identificador del objeto remoto a utilizar.
- Un número de operación que describe al método al que se llama.
- Los parámetros formados.

En el lado del servidor hay otro objeto *stub* (la terminología Java lo denomina *skeleton*), el cual se encarga de tomar la información contenida en paquete que envió el cliente y pasarla al objeto real que ejecuta el método remoto. Las acciones que realiza el *stub* del servidor para cada invocación remota de método son:

- Extraer los parámetros.
- Llamar al método deseado en el objeto remoto real.
- Capturar el valor devuelto o la excepción de la llamada realizada.
- Formar el valor de vuelta.
- Enviar un paquete con el valor de vuelta al *stub* del cliente.

Lo que resta es que el *stub* del cliente extraiga el valor o la excepción del paquete de respuesta recibido del *stub* del servidor, y lo reenvie al cliente.

La sintaxis para una llamada remota es la misma que para una llamada local:

```
// cliente.java = Programa cliente
...
      // Pone el contador al valor inicial 0
      System.out.println("Poniendo contador a 0");
      micontador.sumar(0);
...
```

El cliente siempre utiliza variables cuyo tipo es una interfaz para acceder a los objetos remotos. Por ejemplo, asociado a la llamada anterior estaría la interfaz:

```
// icontador.java = Interfaz para contador
public interface icontador extends java.rmi.Remote {
  int sumar() throws java.rmi.RemoteException;
  void sumar(int valor) throws java.rmi.RemoteException;
  public int incrementar() throws java.rmi.RemoteException;
}
```

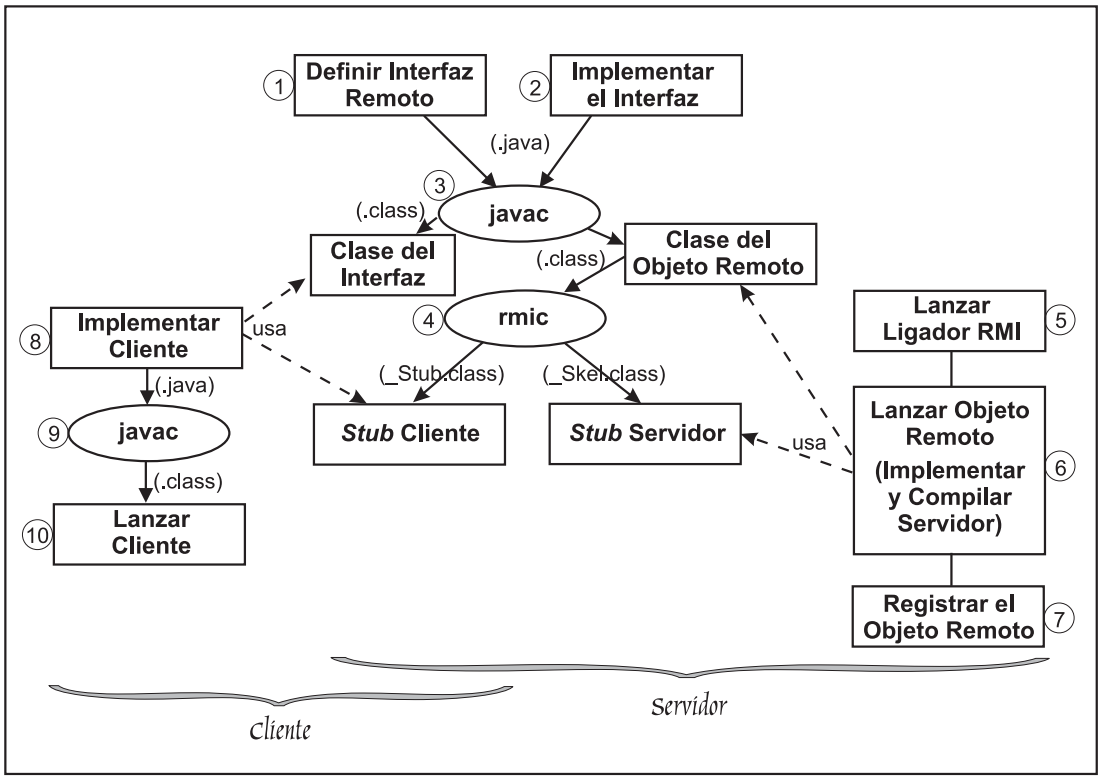
Y la declaración de un objeto para una variable que implementa una interfaz:

```
// cliente.java = Programa cliente
...
// Crea el stub para el cliente especificando el nombre del servidor
      icontador micontador = (icontador)Naming.lookup("rmi://"
                              + args[0] + "/" + "mi contador");
...
```

Por supuesto, las interfaces son entidades abstractas que sólo anuncian que métodos pueden ser llamados junto con sus *signatures*. Variables cuyo tipo sea una interfaz siempre tienen que estar ligadas a un objeto real de algún tipo, en el caso de objetos remotos es una clase *stub*. El programa cliente no conoce realmente el tipo de los objetos remotos. Las clases *stubs* y los objetos asociados se crean automáticamente.

3.2.3 Proceso de Desarrollo RMI

A continuación se muestran y describen los pasos a seguir para crear las clases RMI (cliente y servidor) y ejecutarlas (véase la siguiente figura):



1) **Definir la interfaz remota**. El objeto remoto tiene que declarar sus servicios por medio de una interfaz remota. Esto se hace derivando de la interfaz *java.rmi.Remote*. Cada método en una interfaz remota tiene que lanzar una excepción *java.rmi.RemoteException*.

2) **Implementar la interfaz remota**. Se tiene que proporcionar una clase para el objeto remoto que implemente la interfaz anterior. Esta clase tiene que derivar de *java.rmi.UnicastRemoteObject*.

3) **Compilar la clase del objeto remoto**. Se realiza con *javac*.

4) **Generar los *stubs***. RMI proporciona un compilador denominado *rmic* que genera los *stubs* a partir de la clase remota ya compilada (archivo *.class* resultante del paso anterior). El *stub* del cliente para ese objeto remoto tendrá el sufijo *_Stub.class*, y el del servidor *_Skel.class*.

5) **Lanzar el ligador RMI en el servidor**. RMI define interfaces para un servicio de denominación (ligadura) no permanente llamado *rmiregistry*. Permite registrar y localizar objetos remotos utilizando nombres simples. Cada proceso servidor puede tener un ligador propio, o puede haber un único ligador para todas los programas Java que se estén ejecutando en un mismo computador. Por defecto el ligador tiene asociado el puerto 1099, si se desea otro habrá que especificarlo como parámetro en el momento de lanzarlo, como se muestra a continuación:

```
rmiregistry 1311 \&
```

6) **Lanzar el objeto remoto**. Se tienen que cargar las clases del objeto remoto y después crear las instancias que se deseen. Cuando se desarrolle una aplicación en la cual puedan existir invocaciones remotas a métodos de diferentes objetos que pueden o tienen que formar parte de un único proceso servidor, entonces el diseño a seguir para la aplicación será similar al del ejemplo mostrado más adelante. En éste, el objeto remoto se lanza instanciándolo dentro de un proceso servidor, el cual es un programa normal Java.
7) **Registrar el objeto remoto en el ligador**. Se tienen que registrar todas las instancias de objetos remotos proporcionando un nombre, para que puedan localizarse por los clientes. Para ello se utilizan los métodos de la clase *java.rmi.Naming*. Esta clase utiliza el ligador para almacenar los nombres. Una vez ejecutado este paso, el objeto remoto ya está preparado para ser invocado por los clientes.
8) **Escribir el código del cliente**. Normalmente el cliente será código normal Java, sólo que hay que utilizar también la clase del paso anterior para localizar el objeto remoto.
9) **Compilar el código cliente**. Se utiliza *javac*.
10) **Lanzar el cliente**. Se tienen que cargar las clases del cliente y sus *stubs*.

La siguiente macro se puede utilizar para compilar y ejecutar en una única máquina el ejemplo que se verá más adelante. Observe como realmente la compilación de todos los archivos de programas y clases se lleva a cabo consecutivamente, es decir los pasos 8 y 9 anteriores se pueden llevar a cabo en cualquier momento entre los pasos 2 y 10.

```
#!/bin/sh -e
# ejecutar = Macro para compilacion y ejecucion del programa ejemplo
#            del contador en una sola maquina Unix de nombre localhost.

# Mata todos los procesos
matar() { kill 0

# Configura traps para que interrupciones maten los procesos
trap matar INT EXIT

echo
echo "Lanzando el ligador de RMI"
rmiregistry 1311 &

echo
echo "Compilando con javac ..."
javac *.java

echo
```

```
echo "Creando stubs con rmic ..."
rmic contador

echo
echo "Lanzando el servidor"
java servidor &

sleep 5

echo
echo "Lanzando el cliente"
echo
java cliente turing:1311
```

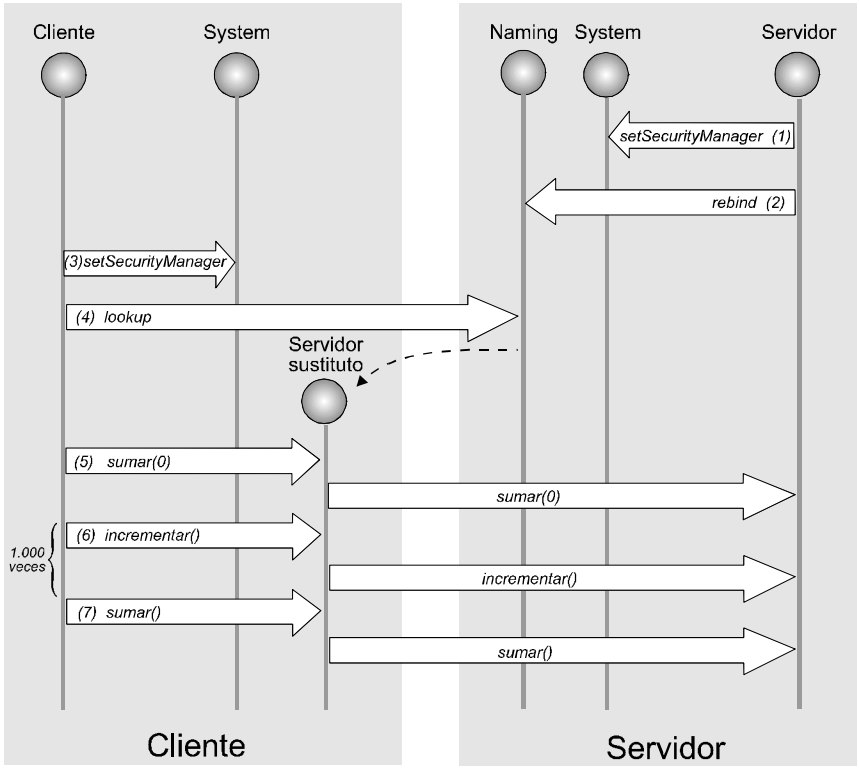
3.2.4 Ejemplo

El programa contador que se muestra a continuación es un pequeño ejemplo de programa cliente-servidor. En este ejemplo el servidor (*servidor.java*) exporta los métodos contenidos en la interfaz *icontador.java* del objeto remoto instanciado como *micontador* de la clase definida en *contador.java*.

El programa cliente es un programa normal Java que realiza las siguientes acciones:

1) Pone un valor inicial en el contador del servidor.
2) Invoca el método *incrementar* del contador 1.000 veces.
3) Imprime el valor final del contador junto con el tiempo de respuesta medio calculado a partir de las invocaciones remotas del método *incrementar*.

La siguiente figura muestra las interacciones RMI entre cliente y servidor:



A continuación se da un listado del código fuente del programa completo:

```
*****************************************
// icontador.java = Interfaz para contador

public interface icontador extends java.rmi.Remote {
  int sumar() throws java.rmi.RemoteException;
  void sumar(int valor) throws java.rmi.RemoteException;
  public int incrementar() throws java.rmi.RemoteException;
}

*********************************************
// contador.java = Implementacion del contador

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class contador extends UnicastRemoteObject implements icontador {
  private int suma;

  public contador(String nombre) throws RemoteException {
    super();
    try {
      // Crea un registro del objeto en el ligador dando un nombre y
      // la referencia al objeto
      Naming.rebind("//turing:1311/" + nombre, this);
```

```
      suma = 0;
    } catch (Exception e) {
      System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
    }
  }

  public int sumar() throws RemoteException {
    return suma;
  }

  public  void sumar(int valor) throws RemoteException {
    suma = valor;
  }

  public int incrementar() throws RemoteException {
    suma++;
    return suma;
  }
}

**********************************
// servidor.java = Programa servidor

import java.rmi.*;
import java.rmi.server.*;

public class servidor {
  public static void main(String args[]) {
    // Crea e instala el gestor de seguridad
    System.setSecurityManager(new RMISecurityManager());

    try {
      // Crea una instancia de contador
      contador micontador = new contador("mi contador");

      System.out.println("Servidor del contador preparado");
    } catch (Exception e) {
      System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

```
**********************************
// cliente.java = Programa cliente

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class cliente {
  public static void main(String args[]) {
    // Crea e instala el gestor de seguridad
    System.setSecurityManager(new RMISecurityManager());

    try {
      // Crea el stub para el cliente especificando el nombre del servidor
      icontador micontador = (icontador)Naming.lookup("rmi://"
                           + args[0] + "/" + "mi contador");

      // Pone el contador al valor inicial 0
      System.out.println("Poniendo contador a 0");
      micontador.sumar(0);

      // Obtiene hora de comienzo
      long horacomienzo = System.currentTimeMillis();

      // Incrementa 1000 veces
      System.out.println("Incrementando...");
      for (int i = 0 ; i < 1000 ; i++ ) {
        micontador.incrementar();
      }

      // Obtiene hora final, realiza e imprime calculos
      long horafin = System.currentTimeMillis();
      System.out.println("Media de las RMI realizadas = "
                       + ((horafin - horacomienzo)/1000f)
                       + " msegs");
      System.out.println("RMI realizadas = " + micontador.sumar());
    } catch(Exception e) {
      System.err.println("Exception del sistema: " + e);
    }
    System.exit(0);
  }
}
```

3.3 Ejercicios

Los siguientes ejercicios pretenden que el estudiante aprenda a diseñar y programar aplicaciones Clientes-Servidor. Aún siendo casos sencillos de aplicaciones cliente-servidor, las interacciones a resolver entre sus componentes permiten abordar los aspectos más importantes (distribución, centralización, funcionalidad, concurrencia, etc.) a tener en cuenta en el desarrollo de aplicaciones cliente-servidor de dos etapas. Escoja uno de ellos para su diseño e implementación.

3.3.1 Ejercicio 3

Este ejercicio se basa en la idea de utilizar varios procesos para realizar partes de una computación en paralelo.

El programa seguirá el esquema de computación maestro-esclavo, en el cual existen varios procesos trabajadores (esclavos) idénticos y un único proceso que reparte trabajo y reúne resultados (maestro). Este esquema permite utilizar el modelo cliente-servidor de interacción entre procesos, en el cual los esclavos (clientes) llamarán a las operaciones que ofrece el maestro (servidor).

Cada esclavo es capaz de realizar cualquiera de los pasos de una computación (incluso varios si así se desea). Un esclavo repetidamente obtiene un trabajo del maestro, lo realiza y devuelve el resultado.

El proceso maestro mantiene un registro de subtrabajos de un trabajo (computación) a realizar. Cada subtrabajo lo realiza uno de los esclavos, y el maestro imprime el resultado que le devuelva el esclavo o lo registra para posteriormente imprimirlo, de manera que la salida de números primos sea ordenada.

El ejercicio concreto a programar es el cálculo de los números primos que hay en un intervalo. Los esclavos calcularán los números primos de cada subproblema (subintervalo que le haya asignado el maestro) aplicando algún método concreto como por ejemplo:

Un número n es primo si no es divisible por algún k tal que $2 < k < \text{sqrt}(n)$

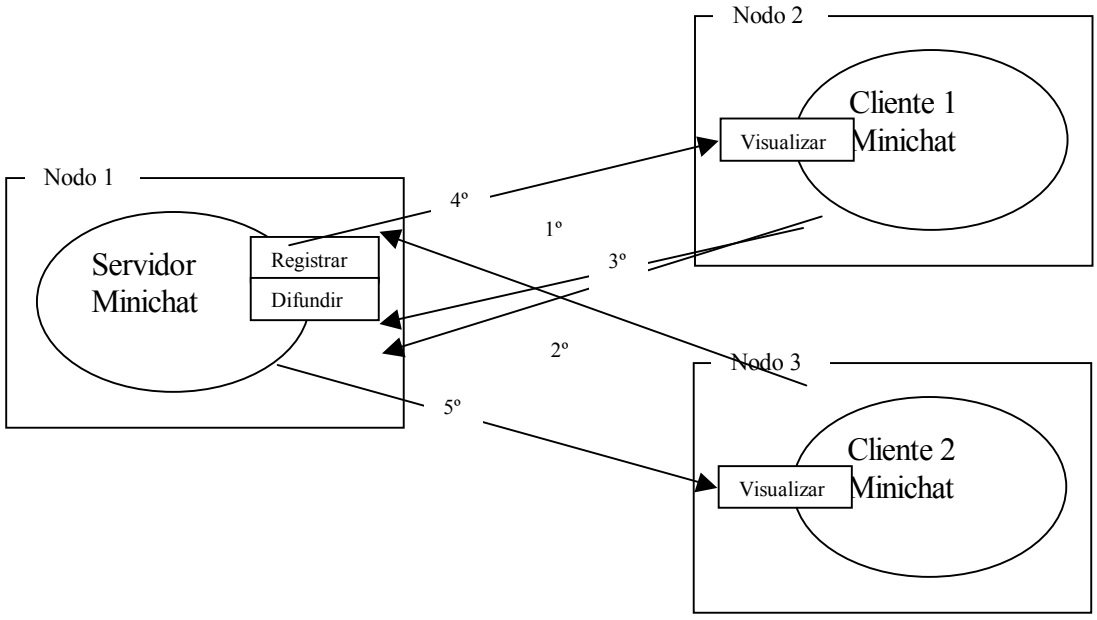
3.3.2 Ejercicio 4

Se plantea como desarrollo un caso simple de mensajería síncrona (*chat*) en una versión centralizada para la difusión de mensajes.

Esta aplicación consta de un servidor centralizado encargado de la difusión de cada mensaje emitido por un cliente a todos los clientes actualmente disponibles en el sistema, incluido el mismo emisor. Por lo tanto, el servidor también será el encargado de llevar un registro centralizado de los clientes que se encuentran disponibles en un momento dado.

Por otro lado, cada cliente, en primer lugar se registrará en el servidor centralizado, y a partir de ese momento repetidamente se encargará de capturar los mensajes de cada usuario que se han de difundir en el sistema por medio del servidor. Además, cada cliente visualizará los mensajes que el servidor centralizado difunde, incluidos los que se generaron en el propio cliente.

La siguiente figura muestra el esquema para este sistema, incluyendo las operaciones básicas (mínimas) que deben exportar cada uno de los componentes. Asimismo, se incluye el patrón general de interacción, en base a la ordenación del intercambio de llamadas a métodos entre componentes, para dicho sistema.

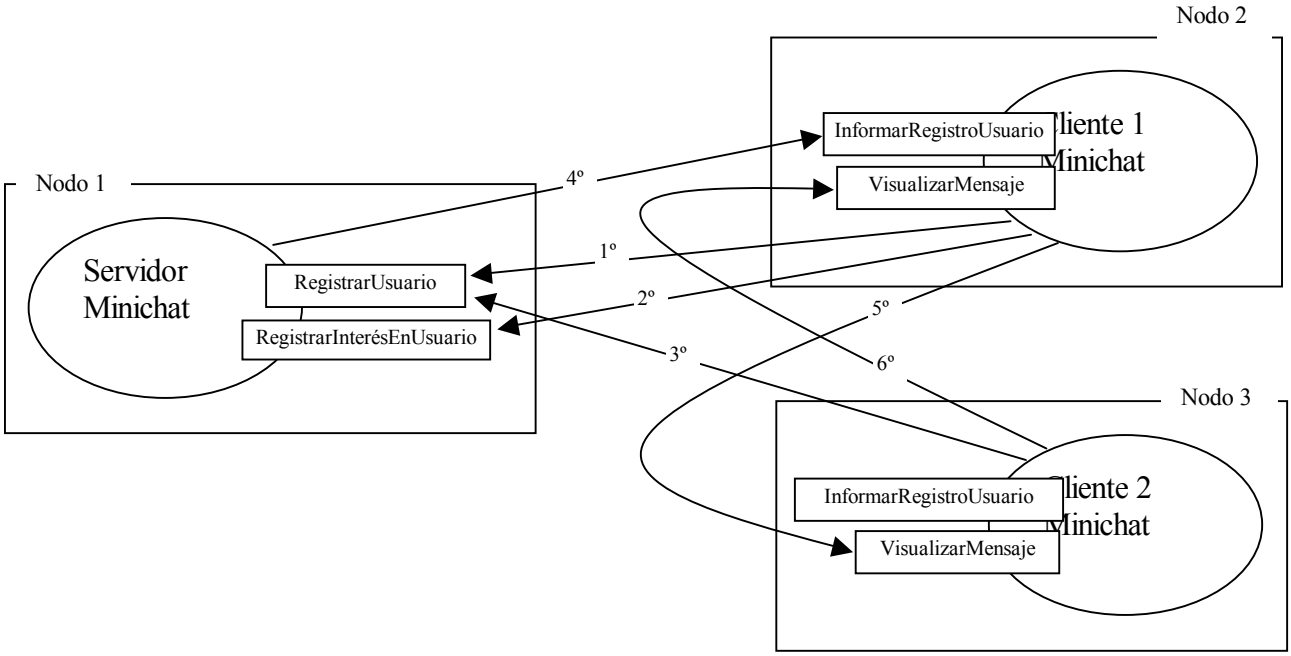


3.3.3 Ejercicio 5

Consiste en una versión distribuida para la comunicación del ejercicio anterior. Esta segunda versión al igual que la versión anterior, consta de un servicio centralizado para mantener un registro de los clientes actualmente disponibles, así como del interés que tiene cada uno de ellos en comunicarse con otros clientes potenciales del sistema. A diferencia de la versión anterior, la comunicación de mensajes propios del sistema *chat* ocurre directamente entre parejas de clientes (comunicación punto a punto).

Para que dicha comunicación se pueda hacer efectiva, los dos clientes tienen que registrarse previamente en el servidor y al menos uno de ellos debe de haber registrado su interés en mantener una comunicación con el otro. Una vez que se dan estos dos hechos, el que haya registrado su interés de comunicación con el otro, puede iniciar la comunicación seleccionando a dicho usuario de una lista privada de usuarios registrados con los cuales tiene interés de mantener una posible comunicación. Por lo tanto, el servidor notificará a cada cliente, cuando otro cliente por el cual ha mostrado interés, se registra en el sistema añadiéndolo de este modo a dicha lista.

Una vez que cualquiera de los dos usuarios cierre la ventana correspondiente a la comunicación entre ellos, ésta concluirá. La siguiente figura muestra el esquema para este sistema, incluyendo las operaciones básicas (mínimas) que deben exportar cada uno de los componentes. Asimismo, se incluye el patrón general de interacción, en base a la ordenación del intercambio de llamadas a métodos entre componentes, para dicho sistema.



4 CORBA

4.1 Introducción

VisiBroker para Java es un completo ORB (*Object Request Broker*) conforme a la especificación CORBA 2. Soporta un entorno de desarrollo para construir, desplegar y gestionar aplicaciones de objetos distribuidos que interoperan a través de plataformas. Las aplicaciones construidas son accesibles por aplicaciones basadas en tecnología Web que se comunican usando el protocolo estándar entre objetos distribuidos IIOP (*Internet Inter-ORB Protocol*) de OMG (*Object Management Group*).

El ORB de Visibroker conecta un programa cliente (el cual puede ser un *applet* o una aplicación estándar Java), ejecutándose en un navegador que soporte Java o en una máquina virtual Java, con los objetos que desee utilizar. El programa cliente no necesita conocer donde residen los objetos, sólo necesita conocer el nombre o la referencia al objeto y comprender como se utiliza su interfaz.

El ORB se encarga de localizar el objeto, enviar la petición y devolver el resultado. El ORB no es un proceso independiente, sino una colección de objetos Java y recursos de red que se integran con las aplicaciones del usuario.

4.2 Características

Visibroker para Java tiene las siguientes características:

- **Repositorio de Interfaces**. El repositorio de interfaces (*Interface Repository*) es una base de datos distribuida en línea. Contiene meta-información acerca de tipos de objetos del ORB. Esta información incluye módulos, interfaces, operaciones, atributos y excepciones. Es un servicio el cual tiene que ser iniciado, y en el que se puede añadir o extraer información.

- **Interfaz Dinámico de Invocación**. El DII (*Dynamic Invocation Interface*) permite que programas clientes puedan obtener información en el repositorio de interfaces sobre una interfaz de un objeto, y construir dinámicamente peticiones para actuar sobre el objeto.

- **Interfaz Dinámica del *Stub* del Servidor**. El DSI (*Dynamic Skeleton Interface*) permite a los servidores enviar peticiones de clientes a objetos que no fueron definidos estáticamente (en

tiempo de compilación). Proporciona un mecanismo para crear la implementación de un objeto que no hereda del *stub* del servidor generado, permitiendo que el propio objeto se registre en el ORB, reciba peticiones de operaciones realizadas por clientes, procese las peticiones, y devuelva el resultado al cliente. Todo esto se realiza de forma transparente al cliente, ya que éste no necesita llevar a cabo nada especial para implementaciones de objetos que utilizan DSI.

- **Ligadura inteligente**. Se escoge el mecanismo de transporte óptimo cuando un cliente obtiene la referencia de un objeto remoto. Así, si la el objeto es local entonces el cliente realiza una llamada local a método, y si es remoto entonces el cliente utiliza IIOP.

- **Agentes Inteligentes**. Es una extensión a la especificación CORBA que facilita obtener referencias a objetos. Proporcionan un servicio de directorio distribuido. Un agente inteligente puede reconectar automáticamente una aplicación cliente al servidor de objetos apropiado si el que se está utilizando actualmente no está disponible debido a un fallo. Los agentes inteligentes además pueden utilizar el demonio de activación de objetos (OAD) para lanzar a demanda instancias de un proceso servidor.

- **Demonio de Activación de Objetos**. Para activar automáticamente un servidor cuando un cliente obtiene la referencia a un objeto, se puede registrar la implementación del objeto con el OAD (*Object Activation Deamon*). El OAD incluye órdenes (como utilidades) y métodos en su interfaz para registrar, eliminar y listar objetos.

- **Gestión mejorada de hebras y conexiones**. Se proporcionan dos políticas de hebras (grupo de hebras o hebra por sesión). Cuando se selecciona una de estas dos políticas para un servidor de objetos se selecciona automáticamente la forma más eficiente para gestionar conexiones entre clientes y servidores.

- **Servicio de Localización**. Es una extensión a la especificación CORBA que proporciona facilidades de propósito general para localizar instancias de objetos. Trabajando con los agentes inteligentes, este servicio puede ver todas las instancias disponibles de un objeto de las cuales un cliente puede obtener la referencia. La principal misión del servicio es el equilibrado de carga.

- **Depurador de peticiones de objetos**. Mediante un GUI Java, este depurador sigue una invocación de una operación (una petición de objeto) de un programa cliente a la implementación de un objeto servidor. También permite poner puntos de ruptura.

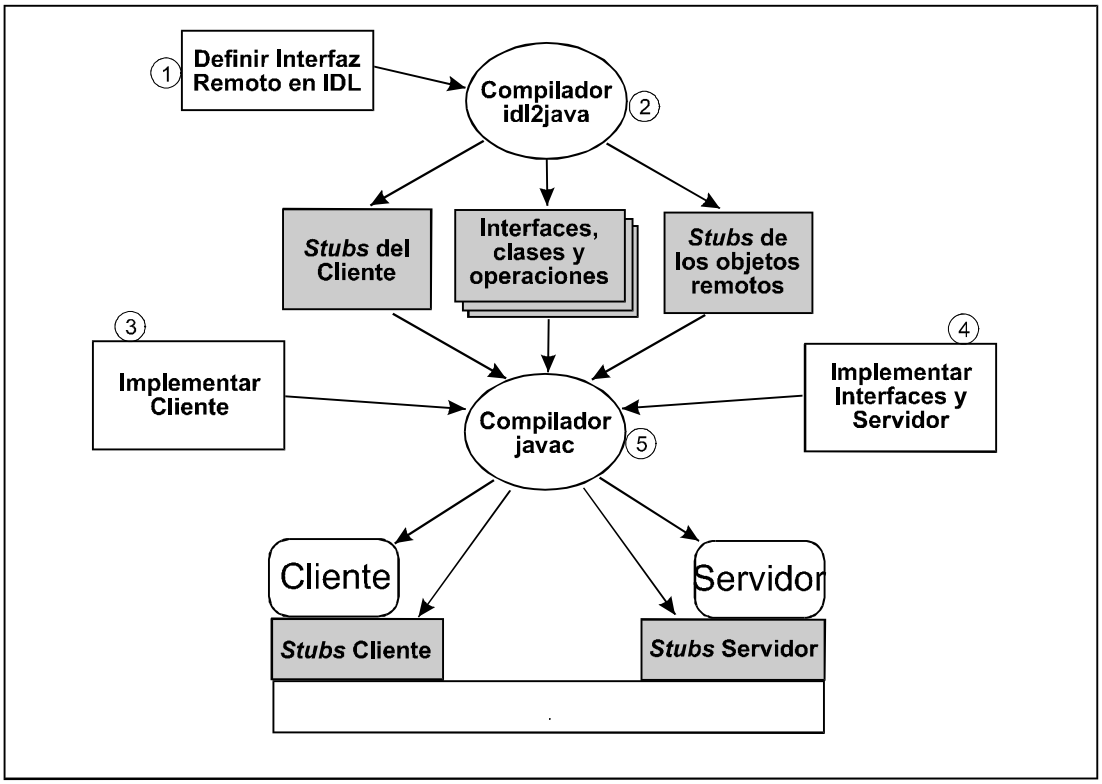
- **Utilidades conocidas como Cafeina**. Incluye:

- **Denominación Web**. Permite asociar URLs (*Uniform Resource Locators*) con objetos, permitiendo que una referencia a un objeto sea obtenida, y un objeto sea contactado, especificando una URL.

- Definición de interfaces sin IDL. El compilador *javaiiop* permite usar el lenguaje Java para definir interfaces. Este compilador genera todo lo necesario (*stubs*, contenedores intermedios de parámetros,...) a partir de código Java. Se puede utilizar para adaptar código Java ya existente a CORBA, o si no se desea aprender IDL. El compilador *java2idl* convierte código Java a IDL, permitiendo así generar código para cualquier otro lenguaje de programación, y volver a implementar objetos Java que soportan el mismo IDL en otros lenguajes.

4.3 Proceso de desarrollo

El siguiente esquema muestra las principales fases en el desarrollo de un aplicación CORBA:



A continuación se describen con detalle los pasos a seguir a modo de tutorial para el desarrollo y ejecución de una aplicación CORBA. Se utiliza el mismo ejemplo contador de la práctica Java anterior.

4.3.1 Creación de un proyecto

Para la creación de un nuevo proyecto lleve a cabo los siguientes pasos:

1) Seleccione *Abrir|Nuevo proyecto*.
2) Sustituya la subcadena *sintitulo1\sintitulo1.jpr* en el valor del campo *Archivo* por *contador\contador.jpr*.
3) Se pueden completar el resto de campos de la ventana.
4) Pulse *Finalizar*.

En el panel de desplazamiento aparecerá el archivo *contador.html* que contiene la información introducida en el asistente de proyecto.

4.3.2 Definición de la interfaz del objeto remoto en Java

En este pequeño tutorial en lugar de definir la interfaz remota directamente con el lenguaje IDL, se realizará en el propio lenguaje Java. El paso siguiente a éste, muestra como obtener el archivo IDL correspondiente utilizando el compilador *java2idl*. Este paso es parte de la fase 1 del esquema de desarrollo.

Para crear el archivo e introducir el código Java que define la interfaz realice lo siguiente:

1) Seleccione *Archivo|Abrir*.
2) Escriba *icontador.java* en el campo *Nombre de archivo*.
3) Active la opción *Añadir al proyecto*.
4) Pulse *Abrir* y seleccione *Sí* cuando pregunte si desea crear el archivo.
5) Escriba en la ventana *Fuente* de ese archivo el siguiente código Java:

```
// icontador.java = Interfaz para contador
package contador;
public interface icontador extends org.omg.CORBA.Object {
  void inicializar (int valor);
  int consultar();
  int incrementar();
}
```

Observará que el nombre de los métodos es diferente a los de la práctica Java anterior pero se mantendrá la misma funcionalidad que tenían.

4.3.3 Generación de la interfaz IDL a partir de la interfaz Java

Es posible generar interfaces IDL a partir de interfaces Java aunque existen ciertas limitaciones. Se pueden utilizar todos los tipos primitivos de Java, pero sólo se pueden utilizar objetos Java para definir la interfaz si el objeto implementa la interfaz *java.io.Serializable*. Este paso es parte de la fase 1 y fase 2 completa del esquema de desarrollo. Realice los siguientes pasos:

1) Pulse con el botón derecho en la interfaz Java creado antes (archivo *icontador.java*) en el panel de desplazamiento y seleccione Propiedades del archivo fuente java.
2) Active las opciones *Generar IDL* y *Generar interfaz IIOP* en *Configuración de Visibroker*, y pulse *Aceptar*.
3) De nuevo pulse con el botón derecho en *icontador.java* en el panel de desplazamiento y seleccione *Ejecutar Make* para compilar la interfaz y generar los archivos IIOP y de la interfaz IDL.
4) Pulse en el símbolo + que aparece justo al lado izquierdo del archivo *icontador.java* para visualizar los archivos generados:
   - la interfaz IDL (*contador_icontador.idl*),
   - los *stubs* del cliente y servidor (*_st_icontador.java* y *_icontadorImplBase.java* respectivamente),
   - el archivo *icontadorHolder.java* que contiene una clase que proporciona un contenedor intermedio para los parámetros a pasar, y
   - el archivo *icontadorHelper.java* que contiene una clase con funciones de utilidad.

4.3.4 Creación del cliente y servidor

A partir de un archivo IDL, se pueden crear aplicaciones CORBA mediante el generador de aplicaciones. La aplicación distribuida puede estar formada por un servidor CORBA programado en Java con un cliente CORBA también programado en Java, o un cliente HTML. Este paso se corresponde con la fase 3 y parte de la 4 del esquema de desarrollo. Siga los siguientes pasos para crear la aplicación:

1) Pulse con el botón derecho en la interfaz IDL generado antes (*contador_icontador.idl*) en el panel de desplazamiento y seleccione *Crear aplicacion* lo cual lanza el generador de aplicaciones.
2) En la ficha Aplicaciones active la opción *Cliente HTML*. Observe que el cliente Java, el servidor y el monitor del servidor ya están seleccionados por defecto.
3) Seleccione *Archivo|Crear* lo cual abre una ventana de diálogo.
4) Pulse en el botón *Crear* para generar todos los archivos del cliente y servidor.
5) Pulse *Cerrar todo* para salir del generador de aplicaciones.

El archivo *contadorAppGenFileList.html* que se añade al proyecto, contiene la lista de los archivos generados.

4.3.5 Implementación del objeto remoto CORBA

Este paso es parte de la fase 4 del esquema de desarrollo. Para implementar la interfaz CORBA realice lo siguiente:

1) Seleccione el archivo *icontadorImpl.java* contenido en el objeto *icontador.sever* del panel de desplazamiento.
2) Añada y modifique el código de este archivo para que finalmente quede como sigue:

```
package contador.server;

import java.sql.*;
import java.util.*;
import java.math.*;

/**
* Archivo de plantilla
*   InterfaceServerImpl.java.template
* Objeto IDL
*   contador.icontador
*
* Archivo fuente IDL
*   C:/JBuilder3/myprojects/contador_icontador.idl
* Resumen
*   Proporciona implementacion por defecto para el
*   lado servidor de una interfaz CORBA.
*/
```

```
public class icontadorImpl extends contador._icontadorImplBase {
  public static ServerMonitorInterface monitor = null;

  int suma = 0;

  private void init() {
    if (monitor == null) {
      monitor = ServerMonitor.addPage(this, "icontador");
      monitor.showObjectCounter(true);
    }
    monitor.updateObjectCounter(1);
  }

  public icontadorImpl(java.lang.String name, java.lang.String creationParameters) {
    super(name);
    init();
  }

  public icontadorImpl(java.lang.String name) {
    super(name);
    init();
  }

  public icontadorImpl() {
    super("icontador");
    init();
  }

  public void inicializar(int arg0)  {
    ServerMonitor.log("(" + _object_name() + ") icontadorImpl.java inicializar()");
    suma = arg0;
  }

  public int consultar()  {
    ServerMonitor.log("(" + _object_name() + ") icontadorImpl.java consultar()");
    return suma;
  }

  public int incrementar()  {
    ServerMonitor.log("(" + _object_name() + ") icontadorImpl.java incrementar()");
    suma ++;
    return suma;
  }
}
```

4.3.6 Compilación y ejecución de la aplicación

Comprende la fase 5 del esquema de desarrollo. Para ejecutar la aplicación realice los siguientes pasos:

1) Para compilar el proyecto seleccione *Proyecto|Ejecutar Make del* proyecto "contador.jpr" y corrija los errores sintácticos si se producen.
2) Ejecute el *Agente Inteligente* que proporciona un servicio de localización de objetos con tolerancia fallos. El cliente encuentra el servidor por medio de este agente, el cual también permite al servidor anunciar sus servicios. Es conveniente que este agente se ejecute al menos en una computadora de una red local para que las aplicaciones puedan ejecutarse. Para lanzar el Agente Inteligente seleccione *Herramientas|Agente Inteligente de Visibroker*. Cuando esté funcionando aparecerá su icono en la barra de tareas.
3) Ejecute el servidor pulsando con el botón derecho sobre el archivo *contadorServerApp.java* del panel de desplazamiento y a continuación seleccione *Ejecutar*. Lo cual lanzará una máquina virtual Java y una interfaz para el monitor del servidor que se incluyó como opción en el Generador de Aplicaciones.
4) Para ejecutar el cliente Java pulse con el botón derecho sobre el archivo *contadorClientApp.java* del panel de desplazamiento y seleccione *Ejecutar*. En una ventana aparecerá la interfaz de usuario que accede al servidor. Cada vez que se realice una invocación remota sobre el servidor, el monitor del servidor visualizará un mensaje.
5) También puede ejecutar la aplicación (incluso simultáneamente con el cliente Java anterior) mediante un cliente HTML, lo cual implica que el servidor debe de ejecutarse en una máquina con servidor Web. Para ello, ejecute el servidor Web que se creo con el Generador de Aplicaciones pulsando con el botón derecho sobre el archivo *ServletServer.java* y seleccione *Ejecutar*. Así se lanza en un ventana DOS el programa *servletrunner* y dicha ventana visualiza su configuración. También aparecerá una ventana que indica la URL completa para ejecutar la aplicación con el cliente (navegador) Web.

4.3.7 Implementación en Java de cliente y servidor específicos para la aplicación

Siempre se pueden crear aplicaciones CORBA en Java sin necesidad de utilizar el Generador de Aplicaciones. A continuación se muestra el código modificado del cliente de la práctica Java anterior, adaptándolo a la especificación CORBA. Incluya este archivo en el proyecto y ejecútelo como ya se ha indicado anteriormente utilizando el servidor creado por el Generador de

Aplicaciones. No olvide que el Agente Inteligente debe de estar ejecutándose siempre antes para que se pueda ejecutar la aplicación.

```
// cliente.java = Programa cliente

public class cliente {
  public static void main(String args[]) {
    // Inicializa el ORB
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

    // Localiza el contador en el Agente Intelligente, el ORB establece un
    // conexión con el servidor apropiado y si esta tiene exito, devuelve
    // un descriptor al objeto. Este descriptor realmente es una referencia
    // al objeto stub (proxy) que se crea.
    // Liga al cliente a un servidor concreto ya que se especifica un
    // nombre de objeto, lo cual distingue entre multiples instancias de
    // una interfaz. Si no se especificara nombre, el Agente Inteligente
    // devuelve cualquier objeto que se corresponda con la interfaz.
    contador.icontador micontador = contador.icontadorHelper.bind(orb,"icontador");


    // Pone el contador al valor inicial 0
    System.out.println("Poniendo contador a 0");
    micontador.inicializar(0);

    // Obtiene hora de comienzo
    long horacomienzo = System.currentTimeMillis();

    // Incrementa 100 veces
    System.out.println("Incrementando...");
    for (int i = 0 ; i < 100 ; i++ ) {
        micontador.incrementar();
    }

    // Obtiene hora final, realiza e imprime calculos
    long horafin = System.currentTimeMillis();
    System.out.println("Media de las llamadas realizadas = "
                       + ((horafin - horacomienzo)/100f)
                       + " msegs");
    System.out.println("Llamadas realizadas = " + micontador.consultar());
    try {
      Thread.sleep(10000);
    } catch (Exception e) {
    System.out.println("Excepcion: "+ e.getMessage());
```

```
    e.printStackTrace();
    }
    System.exit(0);
  }
}
```

A continuación también se muestran el código para el mismo ejemplo de un sencillo servidor y del objeto remoto que instancia, los cuales se podrían utilizar en lugar del código que crea el generador de aplicaciones:

```
// servidor.java = Programa servidor

public class servidor {
  public static void main(String args[]) {
    // Inicializa el ORB
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

    // Inicializa el BOA y permite especificar su tipo (thread pooling,
    // thread per session, ...)
    org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();

    // Crea una instancia de contador especificando un nombre para el
    // objeto si este va a estar disponible a los clientes a traves del
    // Agente Inteligente. Es este caso la referencia al objeto se conoce
    // como persistente, si no se especificara se conoceria como
    // transitoria y solo las entidades que tienen explicitamente esa
    // referencia pueden invocar sus metodos.
    contador.icontador micontador = new contador.contador("icontador");

    // Exporta (activa) el objeto recien creado de forma que ya puede
    // recibir peticiones de clientes. Para ello, lo registra en el
    // Agente Inteligente a traves del BOA si el objeto es persistente
    // (tambien registra el nombre de la interfaz a su vez), en
    // caso contrario el registro no ocurrira.
    boa.obj_is_ready(micontador);

    System.out.println(micontador + " esta preparado");

    // Espera peticiones de clientes
    boa.impl_is_ready();
  }
}
```

```
// contador.java = Implementacion del contador

package contador;

import java.util.*;

public class contador extends contador._icontadorImplBase{
  private int suma = 0;

  public contador(String nombre) {
    super(nombre);
  }

  public  void inicializar (int valor) {
    suma = valor;
  }

  public int consultar() {
    return suma;
  }

  public int incrementar() {
    suma++;
    return suma;
  }
}
```

4.3.8 Ejercicio 6

Siguiendo los pasos del tutorial anterior, desarrolle una aplicación para el mismo ejercicio escogido en la práctica RMI anterior.