Llamadas a Procedimientos Remotos (RPC)

Alvaro Ospina S

2024

Conceptos que deben estar claros

- ¿Qué es un Middleware?
- ¿Qué tipos o clases de Middlewares hay?
- Diseño de aplicaciones distribuidas dependiendo del tipo de middleware.
- Ejemplo:
  - Con el desarrollo de aplicaciones basadas en middleware de comunicaciones como sockets se requiere tener en cuenta?
    - Debe tener en cuenta las consideraciones del diseño de sistemas distribuidos y protocolos, además, en un MW de invocación remota se debe tener en cuenta:
      - Envío/Recepción de tipos de datos
      - Diferentes tipos de codificación de tipos de datos
      - Identificación de componentes del sistema distribuido (objetos, procedimientos, componentes)
      - Mecanismos de activación de objetos en el servidor.
      - Seguridad en tipos de datos (type Safety)
      - Implementación de mecanismos de sincronización en TCP y UDP

En que consiste el modelo de llamadas remotas

- Permitir que un cliente, pueda realizar llamadas a procedimientos, funciones o métodos en otra parte de la red.
- Para que?
- Que retos enfrente un middleware que soporte este paradigma?

Llamadas a procedimientos remotos RPC

- Nace de la época de la programación procedimental.
- Trata de simular la misma semántica de una invocación local.
- La idea con RPC es hacer llamados a procedimientos localizados en otras máquinas.

Llamadas a procedimientos remotos

- Funcionamiento:
  - Un proceso en una maquina A llama a un proceso en la maquina B, el proceso en A se bloquea mientras que se ejecuta el proceso en B.
  - La información que se intercambia son parámetros y se espera respuesta.
- Problemas:
  - Espacio de direccionamiento diferente
  - Diferentes tipos de datos
  - Fallas

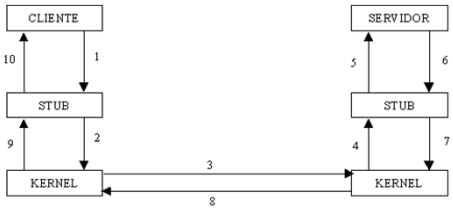
Ejecución de una llamada local

- por ejemplo la llamada invocada desde el *main()*:
  res = sumar(a, b);
- El llamador coloca los parámetros en el stack en orden, último primero.
- se ejecuta la función “sumar”
- se coloca el valor de retorno en un registro, remueve la dir. de retorno y transfiere de nuevo el control al llamador.
- el llamador remueve los parametros del stack y sigue ejecutando otras instrucciones.
- Paso de parámetros:
  - Por VALOR
  - Por REFERENCIA

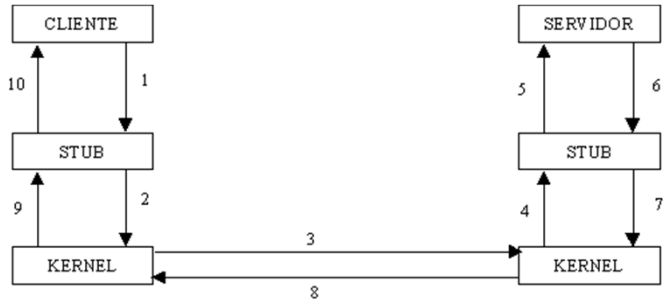
Funcionamiento de RPC

- La idea con RPC es hacer ver a una llamada remota como si fuera local, por esto la invocación debe ser transparente para el que la utiliza.
- La transparencia en RPC se logra agregando un Stub o proxy tanto al cliente como al servidor y utilizando un IDL
- Los pasos que ejecuta un Cliente para invocar un procedimiento remoto en un Servidor son los siguientes:

Funcionamiento de RPC



1. El Cliente llama un procedimiento local llamado el Client Stub (proxy).
2. El Client Stub empaqueta (marshalling) los parámetros, construye un mensaje y lo envia al kernel.
3. El kernel local lo envia al kernel remote donde se encuentra el Server Stub
4. El kernel remoto envía el mensaje al Server Stub.
5. El Server Stub desempaqueta (unmarshalling) los parámetros, identifica el procedimiento y lo ejecuta.
6. El Server Stub recibe el resultado del servidor
7. El Server Stub empaqueta (marshalling) la respuesta, construye un mensaje y lo envia al Kernel.
8. El Kernel remoto lo envia de nuevo al kernel local(cliente).
9. El Kernel local envía el mensaje al Client Stub.
10. Client Stub desempaqueta (marshalling) el resultado y lo retorna de la misma forma que un procedimiento local.

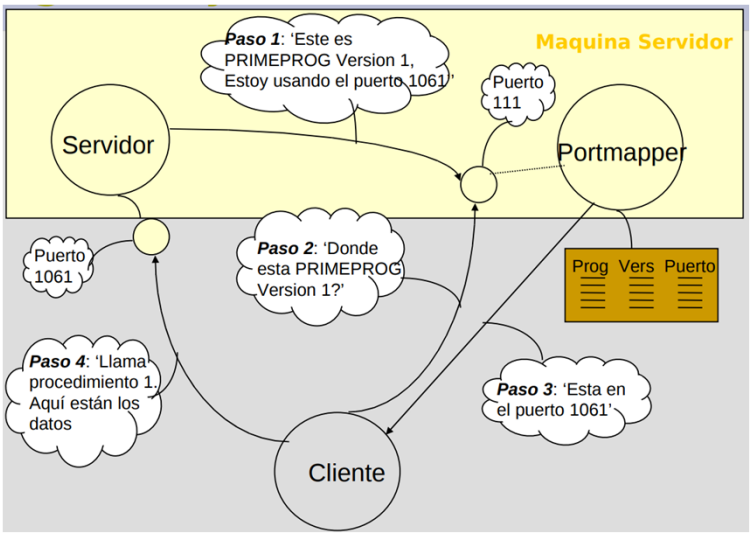


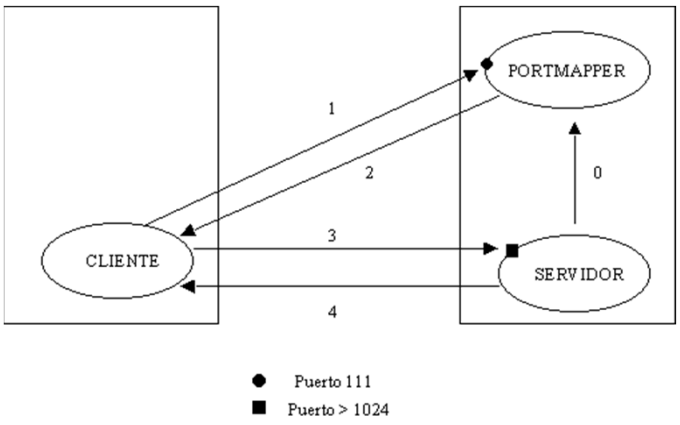
Paso de parámetros en RPC

- Debido a problemas de alfabetos (ASCII/ABCDIC), representación de enteros (complemento a uno o a dos), punto flotante, ordenamiento de bytes (Little Endian y Big Endian), se hace necesario establecer una forma canonica de representar los distintos tipos de datos.
- Se ejecuta un proceso conocido como Marshalling/UnMarshalling, el cual consiste el transformar los tipos de datos locales de una maquina en un formato estándar para ser transmitidos por la red.
- Los tipos básicos como escalares entre otros se pasan por valor, cuando son arreglos tanto en los clientes como servidores se realiza una copia local, se manipulan y se envian por la red.
- En RPC no hay paso de punteros.
- Además de los parámetros normales de una función, se requiere transferir otra información como: Nombre del procedimiento, versión, etc.

Localización de Servidores

- Como un cliente localiza un servidor?
  1) podría ser que el cliente tuviera la dir. del servidor. Esto es muy ineficiente y estático.
  2) realizar un proceso dinámico de asociación o binding con el servidor.
- Partimos de una especificación formal del servidor:
  - nombre del servidor
  - número de versión
  - lista de procedimientos
  - para cada procedimiento se especifica los parámetros
- La especificación formal es muy util para los generadores de código. Cuando un servidor comienza a ejecutar envía un mensaje a un programa (local o remoto) llamado el "binder" o servidor de registro.
- El cliente conoce la dirección del binder a quien interroga por la existencia o no de los servidores que desea contactar.
- Este binder se conoce como un Localizador de Recursos en la red, un broker o un "corredor" de procedimientos.

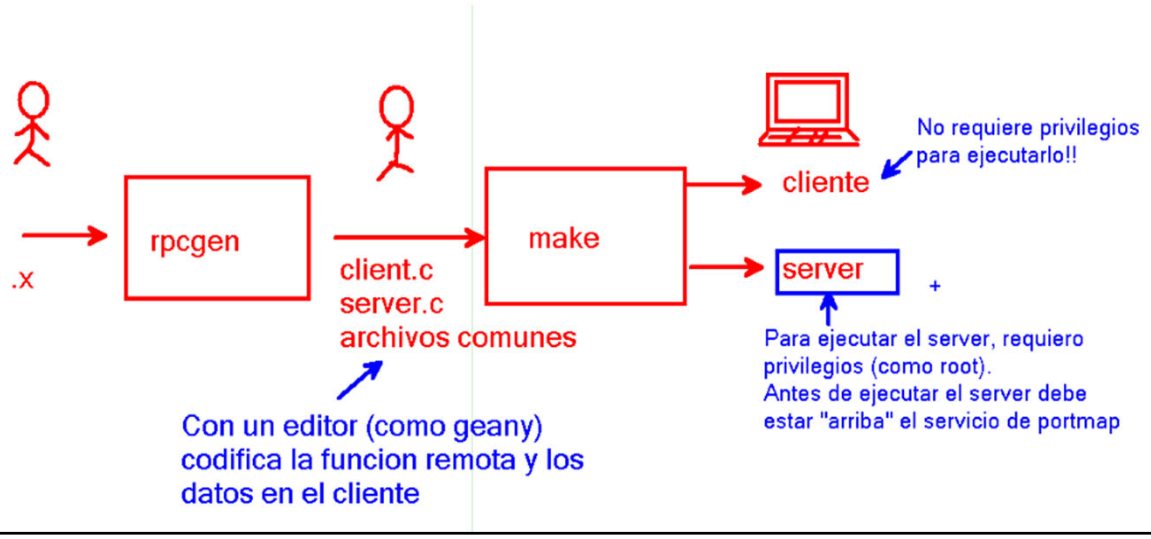




Necesidad de un IDL en RPC

- Se debe separar la especificación de la implementación de los RPC
- Se debe hacer en un lenguaje neutral.
- Para el caso de SUN-RPC existe un lenguaje conocido como RPCL
- En este se especifican los procedimientos
- Se precompilan para generar stubs, proxies, clientes y servidores.

Un ejemplo en RPCL



Un ejemplo en RPCL

archivo.x

```
struct sumandos {
  int sumando1;
  int sumando2;
};

program PROGRAMA_SUMA {
  version VERSION_SUMA {
    int suma(sumandos) = 1;
    int resta(sumandos) = 2;
  } = 1;
} = 0x20000001;
```

Server (sin main)

```
include "sumador.h"

int *
suma_1_svc(sumandos *argp, struct svc_req *rqstp)
{
  static int  result;

  /** insert server code here */
  result = argp->sumando1 + argp->sumando2;

  return &result;
}

int *
resta_1_svc(suamandos *argp, struct svc_req *rqstp)
{
  static int  result;

  /** insert server code here */
  result = argp->suamando1 - argp->sumando2;

  return &result;
}
```

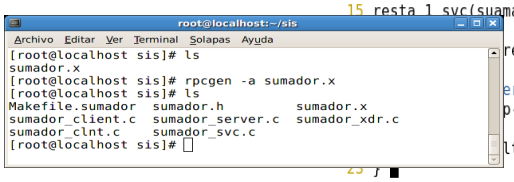
Client

```
#include "sumador.h"


void
programa_suma_1(char *host)
{
    CLIENT *clnt;
    int  *result_1;
    sumandos  suma_1_arg;
    int  *result_2;
    sumandos  resta_1_arg;

    fprintf(stderr, "Cambiando los parametros \n");
   /** Parametros de la suma **/
   suma_1_arg.sumando1 = 5;
   suma_1_arg.sumando2 = 8;

    /** Parametros de la resta **/
    resta_1_arg.sumando1 = 10;
    resta_1_arg.sumando2 = 3;

#ifndef DEBUG
    clnt = clnt_create (host, PROGRAMA_SUMA, VERSION_SUMA, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */
```

Un ejemplo en RPCL

archivo.x

```
1 struct sumandos {
2   int sumando1;
3   int sumando2;
4 };
5
6 program PROGRAMA_SUMA {
7   version VERSION_SUMA {
8     int suma(sumandos) = 1;
9     int resta(sumandos) = 2;
10  } = 1;
11 } = 0x20000001;
12
```



Server (sin main)

```
1 include "sumador.h"
2
3 int *
4 suma_1_svc(sumandos *argp, struct svc_req *rqstp)
5 {
6   static int  result;
7
8   /** insert server code here */
9   result = argp->sumando1 + argp->sumando2;
10
11  return &result;
12 }
13
14 int *
15 resta_1_svc(suamandos *argp, struct svc_req *rqstp)
16 {
17  static int  result;
18
19  /** insert server code here */
20  result = argp->suamando1 - argp->sumando2;
21
22  return &result;
23 }
```

Client

```
#include "sumador.h"

void
programa_suma_1(char *host)
{
    CLIENT *clnt;
    int  *result_1;
    sumandos  suma_1_arg;
    int  *result_2;
    sumandos  resta_1_arg;

    fprintf(stderr, "Cambiando los parametros \n");
    /** Parametros de la suma **/
    suma_1_arg.sumando1 = 5;
    suma_1_arg.sumando2 = 8;

    /** Parametros de la resta **/
    resta_1_arg.sumando1 = 10;
    resta_1_arg.sumando2 = 3;

#ifndef DEBUG
    clnt = clnt_create (host, PROGRAMA_SUMA, VERSION_SUMA, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */
```


Un ejemplo en RPCL

archivo.x

```
1 struct sumandos {
2   int sumando1;
3   int sumando2;
4 };
5
6 program PROGRAMA_SUMA {
7   version VERSION_SUMA {
8     int suma(sumandos) = 1;
9     int resta(sumandos) = 2;
10  } = 1;
11 } = 0x20000001;
12
```

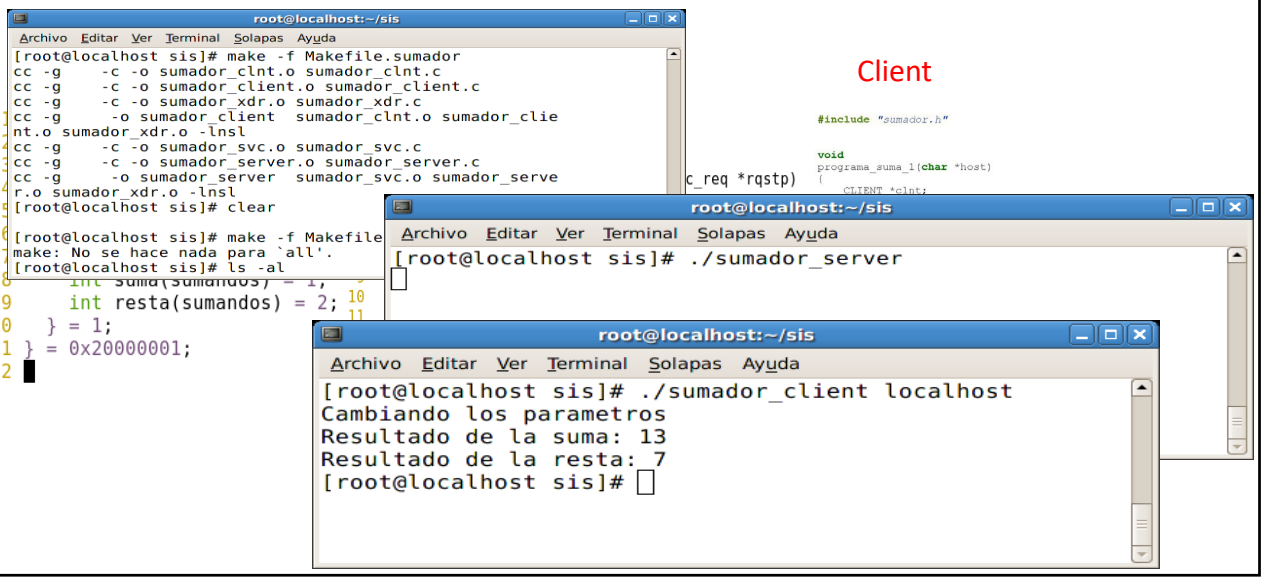
Server (sin main)

```
1 include "sumador.h"
2
3 int *
4 suma_1_svc(sumandos *argp, struct svc_req *rqstp)
5 {
6   static int  result;
7
8   /** insert server code here */
9   result = argp->sumando1 + argp->sumando2;
10
11  return &result;
12 }
13
14 int *
15 resta_1_svc(suamandos *argp, struct svc_req *rqstp)
16 {
17  static int  result;
18
19  /** insert server code here */
20  result = argp->suamando1 - argp->sumando2;
21
22  return &result;
23 }
```

Client

```
#include "sumador.h"

void
programa_suma_1(char *host)
{
    CLIENT *clnt;
    int  *result_1;
    sumandos  suma_1_arg;
    int  *result_2;
    sumandos  resta_1_arg;

    fprintf(stderr, "Cambiando los parametros \n");
    /** Parametros de la suma **/
    suma_1_arg.sumando1 = 5;
    suma_1_arg.sumando2 = 8;

    /** Parametros de la resta **/
    resta_1_arg.sumando1 = 10;
    resta_1_arg.sumando2 = 3;

#ifndef DEBUG
    clnt = clnt_create (host, PROGRAMA_SUMA, VERSION_SUMA, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */
```

Un ejemplo en RPCL



Un ejemplo en RPCL