Lección 9
Las RPC de Sun Microsystems

1ª parte: Introducción y uso de rpcgen

Las RPC de Sun Microsystems

- Las RPC de Sun Microsystems fueron una de las primeras implementaciones comerciales de RPC.

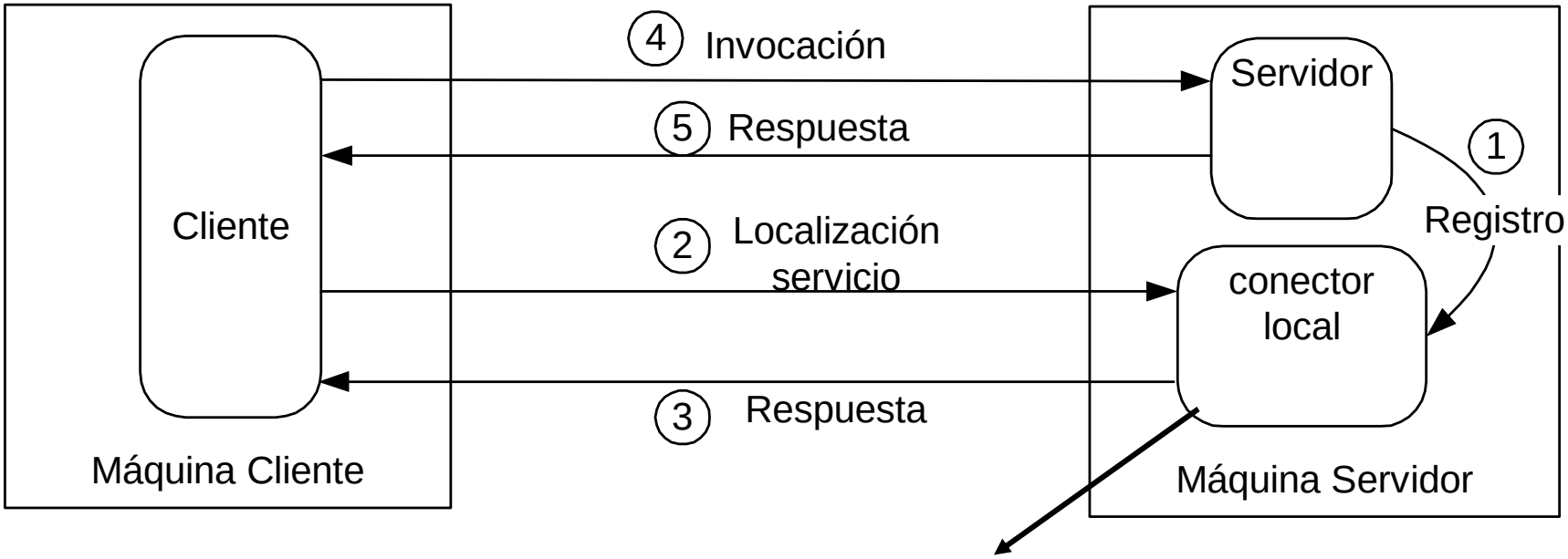
- Actualmente dos versiones:

  ❖ La original.- Presente en la mayoría de sistemas operativos. Código fuente disponible en internet desde 1988.

  ❖ TI-RPC ("*Transport Independent- RPC*"). Sólo en sistemas operativos de SUN (Solaris y SunOS).

Localización e Identificación

Sun utiliza un localizador local



Exige conocer la máquina en la que se encuentra el servicio

Diversos nombres:

- **`Portmapper;`**
- **`rpc.portmap; portmap`**
- **`rpcbind`**

Localización e identificación

La identificación se realiza utilizando números. Cada procedimiento remoto se identifica usando tres cantidades:

- Número de programa al que pertenece el procedimiento. Identifica el servidor al que pertenece. Ejemplo: *Servidor de ficheros.*
- Número de versión del servidor.
- Número de procedimiento. Identifica el servicio concreto dentro del servidor. Ejemplo: *Crear fichero.*

El localizador de Sun almacena sólo los números de programa y versión junto con la información de localización:

Nº Programa	Nº versión	Puerto	protocolo
287340274	2	12348	tcp

Localización e identificación

Restricciones a los valores numéricos:

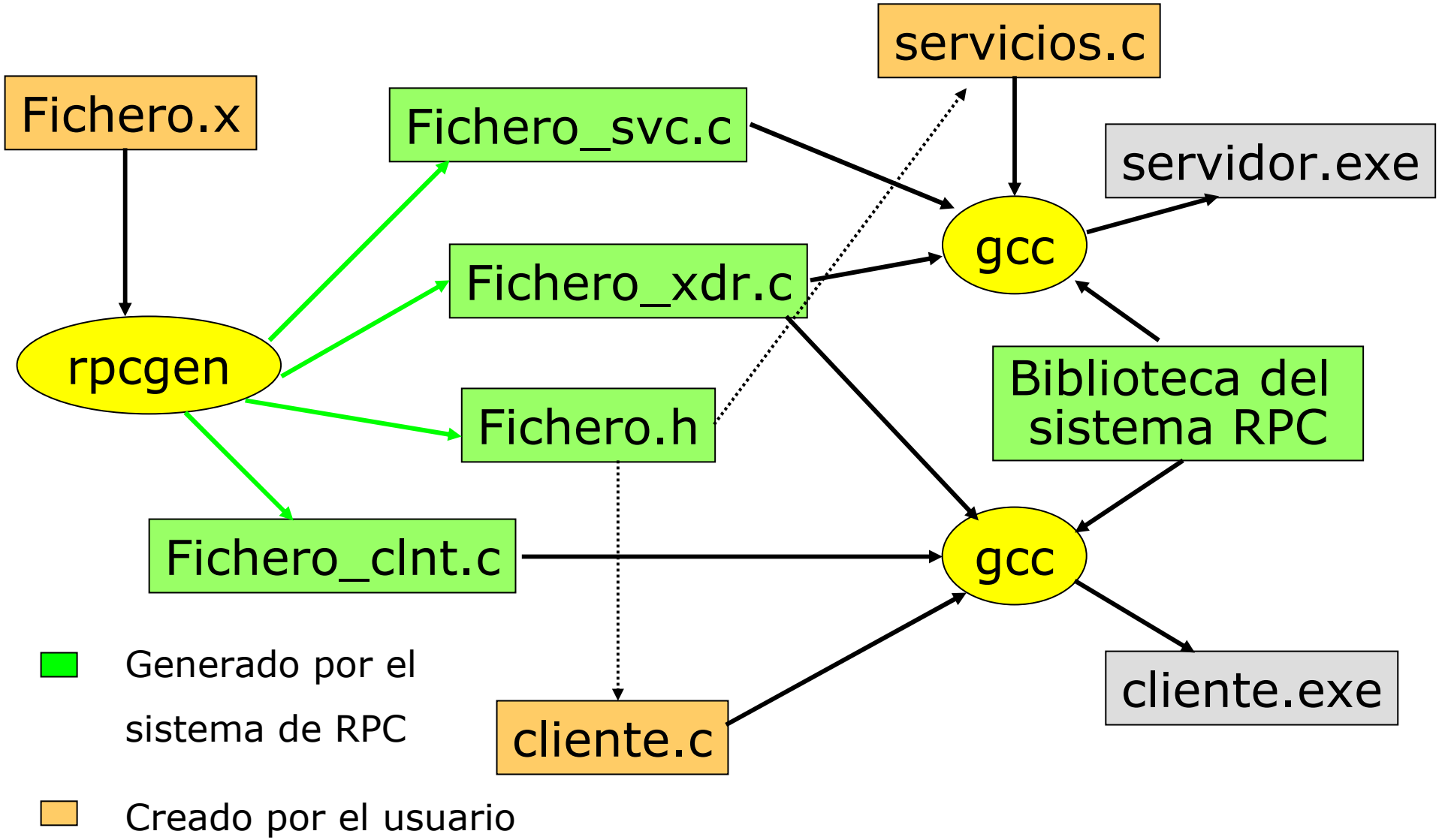
- Versión y Procedimiento: Ninguna.
- Programa:

Nº Programa Rango	Descripción
0x00000000 – 0x1FFFFFFF	Definido por Sun Microsystems
0x20000000 – 0x3FFFFFFF	Definido por el usuario
0x40000000 – 0x5FFFFFFF	Transitorio. Generados dinámicamente por las aplicaciones.
0x60000000 – 0xFFFFFFFF	Reservado

Niveles de utilización

- Utilizando **`rpcgen`**. Más sencilla la programación. Se requiere conocer RPCL (el IDL de Sun). Alto nivel de transparencia.
- Nivel simplificado. Se usan pocas funciones. Se requiere más conocimiento del funcionamiento del sistema de RPC. Transparencia muy baja.

Utilizando RPCGEN



Lenguaje RPCL

El especificación del interface en `Fichero.x` se realiza utilizando el lenguaje RPCL ("*RPC Language*").

RPCL= XDR + extensiones para definir procedimientos

Procedimiento:

```
id_tipo nombre_proc (id_tipo) = valor;
```

Lista de procedimientos:

```
Procedimiento; lista de procedimientos;
```

Versión:

```
version id_version { lista de procedimientos }= valor;
```

Lista de versiones:

```
Versión; lista de versiones;
```

Programa:

```
program nombre_programa { lista de versiones } = valor;
```

Ejemplo RPCL

Interface.x

```
program SERVIDOR_CALCULO {
    version SC_TERCERA {
            int duplica (int) = 1;
            float ENTRE3 (int) = 2;
    } = 3;
} = 0x201ABCBF;
```

Desarrollo con rpcgen.

Interface.h

```
#include <rpc/rpc.h>


#define   SERVIDOR_CALCULO ((unsigned long)(0x201ABCBF))
#define   SC_TERCERA ((unsigned long)(3))
#define   duplica ((unsigned long)(1))
extern  int * duplica_3();
#define   ENTRE3 ((unsigned long)(2))
extern  float * entre3_3();
extern int servidor_calculo_3_freeresult();
```

Desarrollo con rpcgen. Servicios.

servicios.c

```
#include “interface.h”
int * duplica_3 (int *dato, struct svc_req *req)
{
    static int resultado;
    resultado= *dato * 2;
    return(&resultado);
}
float * entre3_3 (int *dato, struct svc_req *req)
{
    static float resultado;
    resultado= (float) *dato/3.0;
    return(&resultado);
}
```

Contiene información sobre el cliente que invoca el servicio.

(Más detalles en próximas lecciones.)

Desarrollo con `rpcgen`. Cliente.

Para construir un cliente mínimo necesitamos utilizar cuatro funciones del API, básicamente de gestión de errores:

`clnt_create`
`clnt_pcreateerror`
`clnt_perror`
`clnt_destroy`

Desarrollo con rpcgen. Cliente.

cliente.c

```
#include "interface.h"

main (int narg, char *args[])
{
  CLIENT *clnt;
  int *resul, dato;
  char *maquina="maquina.servidor.es";

  clnt=clnt_create(maquina,SERVIDOR_CALCULO,SC_TERCERA,udp");

  if (clnt== NULL) {
    clnt_pcreateerror(maquina);
    exit(-1);
  }

  resul=duplica_3 (&dato, clnt);

  if (resul==NULL) {
    clnt_perror(clnt,"Fallo en la llamada:");
    exit(-1);
  }
  printf("\nEl resultado es %d\n", *resul);
  clnt_destroy(clnt);
}
```

Desarrollo con `rpcgen`. Datos complejos

sumador.x

```
struct datos {
    int sum1;
    int sum2;
}
program SUMADOR {
    version SUM_TERCERA {
            int suma (datos) = 1;
            double div3 (float) = 2;
    } = 3;
} = 0x201ABCBF;
```

Datos complejos con `rpcgen`. Servicios.

servicios.c

```
#include “sumador.h”
int * suma_3 (datos *sumandos, struct svc_req *req)
{
    static int resultado;

    resultado= sumandos->dato1 + sumandos->dato2;
    return(&resultado);
}
```

Datos complejos con rpcgen. Cliente.

cliente.c

```
#include “sumador.h"

main (int narg, char *args[])
{
  CLIENT *clnt;
  int *resul;
  datos nums;
  char *maquina="maquina.servidor.es";

  clnt=clnt_create(maquina,SUMADOR,SUM_TERCERA, "udp");

  nums.sum1= 5;
  nums.sum2=235;
  resul=suma_3 (&nums, clnt);

  if (resul==NULL) {
    clnt_perror(clnt,"Fallo en la llamada:");
    exit(-1);
  }
  ................
}
```

Extremo del cliente generado por rpcgen.

sumador_clnt.c

```
#include "sumador.h"
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };
int * suma_3(argp, clnt)
     datos *argp;
     CLIENT *clnt;
{
  static int clnt_res;
  memset((char *)&clnt_res, 0, sizeof (clnt_res));
  if (clnt_call(clnt, SUMA,
           (xdrproc_t) xdr_datos, (caddr_t) argp,
           (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
           TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
  }
  return (&clnt_res);
}
```

Extremo del servidor generado por rpcgen. (I)

sumador_svc.c

```
#include "sumador.h"
int main(int argc, char **argv)
{
  register SVCXPRT *transp;

  (void) pmap_unset(SUMARPROG, SUMARVERS);
  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {  fprintf(stderr, "cannot create udp service."); exit(1);
  }
  if (!svc_register(transp, SUMADOR, SUM_TERCERA, sumador_3, IPPROTO_UDP)) {
    fprintf(stderr, "unable to register (SUMADOR, SUM_TERCERA, udp)."); exit(1);
  }
  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {   fprintf(stderr, "cannot create tcp service."); exit(1);
  }
  if (!svc_register(transp, SUMADOR, SUM_TERCERA, sumador_3, IPPROTO_TCP)) {
    fprintf(stderr, "unable to register (SUMADOR, SUM_TERCERA, tcp).");
    exit(1);
  }
  svc_run();
  fprintf(stderr, "svc_run returned"); exit(1);
}
```

Función *dispatcher* o repartidora

Extremo del servidor generado por rpcgen. (II)

Función *dispatcher*

```
static void sumador_3(struct svc_req *rqstp, register SVCXPRT *transp) {
  union {
    datos suma_1_arg;
    float div3_1_arg;
  } argument;
  char *result;
  xdrproc_t xdr_argument, xdr_result;
  char *(*local)(char *, struct svc_req *);

  switch (rqstp->rq_proc) {
  case NULLPROC:
    (void) svc_sendreply(transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;
  case SUMA:
    xdr_argument = (xdrproc_t) xdr_datos;
    xdr_result = (xdrproc_t) xdr_int;
    local = (char *(*)(char *, struct svc_req *)) suma_3;
    break;
  case DIV3:
    xdr_argument = (xdrproc_t) xdr_float;
    xdr_result = (xdrproc_t) xdr_double;
    local = (char *(*)(char *, struct svc_req *)) div3_3;
    break;
```

Contiene todos los tipos de datos de entrada de los procedimientos

Resultado genérico.

Puntero a una función genérica.

Inicialización para un servicio concreto

..... (sigue)

Extremo del servidor generado por rpcgen. (II)

Función *dispatcher*

```
                .........

    default:
      svcerr_noproc(transp);
      return;
    }

  if (!svc_getargs(transp, xdr_argument, (caddr_t) &argument)) {
      svcerr_decode(transp);
      return;
    }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
      svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument, (caddr_t) &argument)) {
      fprintf(stderr, "unable to free arguments");
      exit(1);
    }
    return;
  }
```

Extracción del parámetro desde la petición.

Invocación del servicio.

Envío de la respuesta.

Opciones de `rpcgen`.

Comunes a la mayoría de las máquinas:

`-c`	Genera únicamente los filtros XDR.
`-h`	Genera únicamente el fichero de cabecera .h
`-l`	Genera únicamente el extremo del cliente.
`-m`	Genera únicamente el extremo del servidor.
`-s [udp|tcp]`	Sólo usa el protocolo especificado.

Para Solaris y Linux:

`-Sc`	Genera un esqueleto de cliente.
`-Ss`	Genera un esqueleto de servidor.
`-Sm`	Genera un fichero `makefile` para la aplicación.
`-a`	Genera todos los ficheros.

Opciones de `rpcgen`.

Cuando `rpcgen` compila están activos una serie de símbolos:

Símbolo	Momento Activo
RPC_HDR	Al generar el fichero `.h`
RPC_XDR	Al generar el fichero `_xdr.c`
RPC_SVC	Al generar el fichero `_svc.c`
RPC_CLNT	Al generar el fichero `_clnt.c`

Además, las líneas que comiencen con el símbolo % pasan directamente al fichero de salida.

Control de la RPC en el cliente: `clnt_control`

```
bool_t clnt_control (
                        CLIENT *clnt,
                        int peticion,
                        char *info
                    );
```

Modifica las características de la estructura de tipo `CLIENT` asociada a la RPC. Necesita que la estructura `clnt` haya sido inicializada con `clnt_create`.

Los valores de `peticion` e `info` dependen de la operación a realizar (siguiente transparencia)

Control de la RPC en el cliente: `clnt_control`

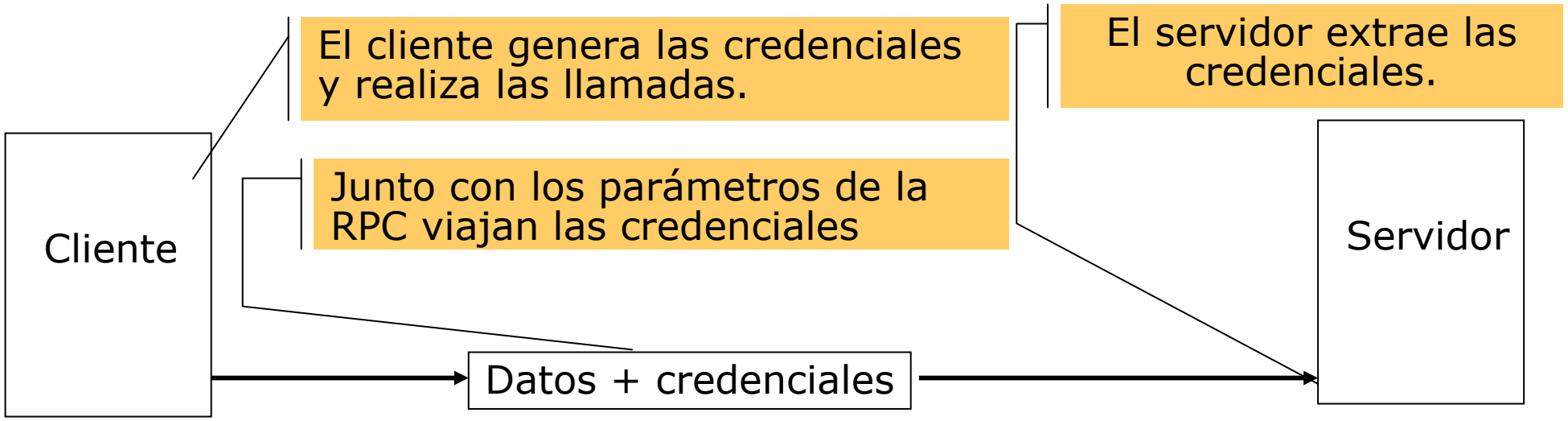
Peticion	info	Accion
`CLSET_TIMEOUT`	`struct timeval`	Fija el tiempo máximo de espera por respuesta del servidor.
`CLGET_TIMEOUT`	`struct timeval`	Obtiene el tiempo máximo de espera activo antes de abortar la llamada.
`CLGET_SERVER_ADDR`	`struct sockaddr_in`	Obtiene los datos de la máquina del servidor.
Solo para estructuras CLIENT inicializadas con UDP		
`CLSET_RETRY_TIMEOUT`	`struct timeval`	Fija el tiempo de espera por respuesta del servidor antes de reintentar la llamada.
`CLGET_RETRY_TIMEOUT`	`struct timeval`	Obtiene el tiempo antes de reintentos activo.

Ejemplo de uso de clnt_control

```
main ()
{
  CLIENT *clnt;
  struct timeval espera;
  .....
  clnt=clnt_create(maquina, DUPLICA,DUPLIVERS,"tcp");
  .......
  espera.tv_sec=40;
  espera.tv_usec=0;
  if (clnt_control(clnt,CLSET_TIMEOUT,&espera)==FALSE) {
    printf("Error");
    exit(-1);  }
  resul=duplica_1 (&dato, clnt);
  ......
}
```

Seguridad

- El sistema de SUN no incorpora ningún tipo de control de acceso a los servicios suministrados vía RPC.
- No se implementa ningún "nivel" de seguridad en el protocolo.
- Cualquier cliente puede invocar cualquier servicio.
- Únicamente se incorpora autentificación del cliente incorporando sus credenciales con cada RPC. Esquema:



Tipos de credenciales

Existe una serie de sistemas de credenciales utilizables con las RPC de Sun:

Identificador	Descripción
AUTH_NONE, AUTH_NULL	Sin credencial. Es el tipo utilizado por defecto.
AUTH_UNIX, AUTH_SYS	Tipo de credencial basado en el sistema operativo UNIX: uid, gid... Sin verificación adicional.
AUTH_SHORT	Similar al anterior, aunque simplificado.
AUTH_DES	Credenciales basadas en el DES *("Data Encryption Standard")*.
AUTH_KERB	Credenciales basadas en el sistema Kerberos desarrollado por el MIT.

Credenciales en el cliente

- El cliente genera sus credenciales llamando a una función específica para cada tipo utilizado.
- Las guarda en la estructura `AUTH` dentro de la estructura `CLIENT`.

```
typedef struct {
        AUTH *cl_auth;
        struct cl_ops {
                .....
} CLIENT;
```

Credenciales cliente

Verificación

Credenciales DES

Operaciones con las credenciales

```
typedef struct {
        struct opaque_auth ah_cred;
        struct opaque_auth ah_verf;
        union des_block ah_key;
        struct auth_ops {
                void (*ah_nextverf)();
                int  (*ah_marshal)();
                int  (*ah_validate)();
                int  (*ah_refresh)();
                int  (*ah_destroy)();
        } *ah_ops;
        caddr_t ah_private;
} AUTH;
```

La estructura opaque_auth

- Esta estructura es la que realmente transporta las credenciales

```
struct opaque_auth {
        enum_t oa_flavor;
        caddr_t oa_base;
        u_int oa_length;
};
```

Valor que identifica el tipo de credencial utilizado (AUTH_UNIX, AUTH_DES, etc)

Puntero a una zona de memoria en la que se encuentran las credenciales.

Tamaño en bytes de esa zona de memoria.

- Cuando un cliente ha terminado de usar unas credenciales puede destruirlas utilizando `auth_destroy`:

```
void auth_destroy ( struct AUTH *auth );
```

Verificación en el servidor

- El servidor recibe una copia de las credenciales del cliente en cada invocación de un servicio.
- El segundo parámetro de cada RPC es una estructura **svc_req**:

```
struct svc_req {
        unsigned long rq_prog;
        unsigned long rq_vers;
        unsigned long rq_proc;
        struct opaque_auth rq_cred;
        caddr_t rq_clntcred;
        SVCXPRT *rp_xprt;
};
```

Datos de la llamada

Credenciales cliente recibidas por la red

Las credenciales procesadas

Datos de la capa de transporte (IP cliente, etc)

- La estructura **rq_cred** es opaca al servidor excepto el campo **oa_flavor** que contiene la identificación del esquema de seguridad utilizado.

Verificación en el servidor (II)

El sistema RPC garantiza:

1. El campo **`rq_cred`** está bien formado y se puede consultar el campo **`oa_flavor`**. Si el esquema de seguridad no está soportado por el ONC RPC, los otros campos de **`rq_cred`** también pueden ser consultados para obtener las credenciales del cliente.

2. Si el esquema no está soportado, **`rq_clntcred`** es un puntero a NULL. Si está soportado, **`rq_clntcred`** es un puntero a una estructura que contiene las credenciales del cliente para ese esquema de seguridad.

```
struct svc_req {
        unsigned long rq_prog;
        unsigned long rq_vers;
        unsigned long rq_proc;
        struct opaque_auth rq_cred;
        caddr_t rq_clntcred;
        SVCXPRT *rp_xprt;
};
```

Seguridad. Ejemplo de uso. Credenciales UNIX

- Las credenciales tipo UNIX de un cliente son las siguientes:

```
struct authunix_parms {
        unsigned long aup_time; /* Fecha creación credencial */
        char *aup_machname; /* Nombre máquina del cliente */
        uid_t aup_uid; /* Uid cliente en esa máquina */
        gid_t aup_gid; /* Gid cliente al crear la credencial */
        unsigned int aup_len; /* tamaño siguiente campo */
        gid_t *aup_gid; /* lista grupos a los que pertenece
};                                  el cliente */
```

- Para crear las credenciales, el cliente invoca la macro:

```
(AUTH *) authunix_create_default ();
```

Ejemplo de uso. Credenciales UNIX. Cliente.

- La secuencia de llamadas sería la siguiente:

```
CLIENT *clnt;
      .......
clnt=clnt_create(....);
clnt->cl_auth=authunix_create_default();
      ......
resultado=llamada1_1(&dato,clnt);
      .....
```

Nota: *el esquema de credenciales tipo unix es muy débil ya que no incorpora ni verificación ni encriptación de datos a través de la red.*

Ejemplo de uso. Credenciales UNIX. Servidor.

- El servidor puede acceder a las credenciales en cada procedimiento:

```
void * acaba_1(void *nada, struct svc_req *req)
{
  struct authunix_parms *ucred;
  if ( req->rq_cred.oa_flavor == AUTH_UNIX ) {
    ucred = ( struct authunix_parms *)req->rq_clntcred;
    if (ucred->aup_uid == MIUID) {
      pmap_unset(PROGNUM, PROGVERS);
      svc_exit();
    }
  } else
      svc_weakauth(req->rq_xprt);
}
```

¿Tipo de credenciales correcto?

Leemos las credenciales

Cliente correcto

Cliente no autorizado

Prototipo svcerr_auth

```
void svcerr_weakauth ( SVCXPRT *xprt );
```

Envía un mensaje al cliente diciendo que sus credenciales son *débiles.*

Es una particularización de:

```
void svcerr_auth (SVCXPRT *xprt, enum auth_stat porq);
```

```
enum auth_stat {
  AUTH_OK = 0,
  AUTH_BADCRED = 1,      /* Credenciales incorrectas */
  AUTH_REJECTEDCRED = 2,  /* El cliente debería iniciar otra sesión */
  AUTH_BADVERF = 3,      /* La verificación es incorrecta */
  AUTH_REJECTEDVERF = 4,  /* El verificador ha expirado o está repetido */
  AUTH_TOOWEAK = 5,      /* Rechazado por razones de seguridad */
  AUTH_INVALIDRESP = 6,          /* Verificador incorrecto de respuesta */
  AUTH_FAILED = 7,       /* Razón desconocida */
};
```

Prototipo `clnt_create`

```
CLIENT * clnt_create (
```

Nombre de la máquina del servidor → `char *host,`

Identificación del servidor:
Nº de programa, nº de versión →
`u_long prognum,`
`u_long versnum,`

Protocolo que se pretende utilizar para acceder al servidor. Es necesario que el servidor se haya registrado con ese protocolo. → `char *protocolo,`

```
);
```

Si existe el servidor en la máquina indicada, retorna un puntero a una estructura de tipo `client` correctamente inicializada.

volver

Prototipo `clnt_pcreateerror`

```
void clnt_pcreateerror ( const char *info );
```

Envía un mensaje a la salida estándar de error indicando porqué la estructura `client` no ha podido ser inicializada. Añade delante del mensaje la cadena `info` seguida de dos puntos.

volver

Prototipo `clnt_perror`

```
void clnt_perror (
                    const CLIENT *clnt,
                    const char *info
                  );
```

Envía un mensaje a la salida estándar de error indicando porqué falló la llamada al procedimiento. Añade delante del mensaje la cadena `info` seguida de dos puntos. Está asociada a una estructura `client`.

volver

Prototipo `clnt_destroy`

```
void clnt_destroy ( CLIENT *clnt );
```

Elimina los recursos asociados a la estructura `clnt`. Después de usar esta función no se puede realizar una llamada a un procedimiento remoto usando la estructura `clnt`.

volver