2. RPC – Apelarea procedurilor la distanţă

RPC (Remote Procedure Call) este o tehnică utilizată pentru construirea aplicaţiilor distribuite bazate pe modelul client-server [Birrel84]. RPC extinde noţiunea de apel local de procedură (funcţie), diferenţa fiind că procedura apelata nu se află în acelaşi spaţiu de adresare cu procedura apelantă. Cele două procese implicate pot sa fie pe acelaşi calculator sau pot sa fie pe doua calculatoare în reţea. Utilizând PRC, programatorii de aplicaţii distribuite ocolesc dezvoltarea interfaţării aplicaţiei cu reţeaua.

Un protocol de transport care suportă paradigma cerere/răspuns este mai mult decât un mesaj UDP care merge într-o direcţie urmat de un alt mesaj UDP care merge în direcţia opusă. De asemenea, nici protocolul TCP nu rezolvă această problemă. Protocolul RPC a fost proiectat în acest scop. El foloseşte aceeaşi semantică a apelului local de procedură, în sensul că dacă un proces face un apel într-o procedură el nu trebuie să-şi pună problema dacă este global sau local.

Faţă de apelul local, apelul la distanţă trebuie să rezolve, în plus două probleme principale:

- apelul la distanţă se face la nivelul unei reţele, fapt care limitează dimensiunea mesajelor. Există riscul de a pierde şi reordona mesajele.
- calculatoarele pe care se execută procesele apelant, respectiv apelat pot avea arhitecturi şi formate de reprezentare a datelor diferite.

Protocolul RPC se află la nivelul Prezentare din stiva OSI şi poate fi considerat la nivelul Aplicaţie în modelul TCP/IP. Versiunea originală de RPC a fost definită în RFC 1050. Versiunea 2 de RPC e definită in RFC 1057, iar versiunea ONC-RPC (Open Network Computing) versiunea 2 e definita în RFC 1831. El a fost propus de Sun Microsystems si acceptat ca standard de facto in lumea UNIX.

În acest capitolul vom discuta aspectele legate de configurarea şi adresarea serviciului de RPC (pe sistemele de operare Linux si Windows), modelul de date folosit (XDR [Sriniv95]) şi două generatoare de cod (rpcgen), modul de realizare a unui client şi a unui server folosind RPC (cu exemple pentru sistemele de operare Linux şi Solaris). Vom analiza modelul XML-RPC propus ca serviciu Java pentru RPC şi vom analiza o aplicaţie de monitorizare bazată pe RPC. Aplicaţia propusă va face apel la noţiunile prezentate pe parcursul acestui capitol.

2.1. Configurarea RPC in Linux şi Windows

Vom prezenta în continuare câteva aspecte cu privire la modul de funcţionare RPC. Vor fi precizate modalităţile de a verifica, porni, opri serviciul de RPC, porturile folosite, modul de identificare a unui server RPC.

2.1.1. RPC in Linux

RPC furnizează un punct central pentru aplicaţiile server sa obţină porturi TCP/UDP pe maşina respectiva pe care rulează. Aplicaţiile nu trebuie *legate* pe anumite porturi specificate pentru că **serviciul de port-mapping** al RPC-ului furnizează porturi libere mai mari de 1024 servere-lor RPC.

Serviciul de lookup folosit include **PORTMAPPER-UL (PMAP)** si **RPCBIND** care sunt descrise in RFC 1833. Portmapper are un port fix pe care stă deschis (portul 111) fie TCP, fie UDP. Acest serviciu de lookup trebuie deschis înaintea serverului/clientului şi trebuie să rămână funcţional pe toată durata execuţiei aplicaţiei RPC.

Server-ele RPC nu utilizează porturi rezervate (aşa cum sunt cele specificate pentru servicii în fişierul */etc/services*); atunci când pornesc, utilizează portul disponibil dat de serviciul **portmapper.** Pentru a vedea care sunt serviciile RPC pornite pe maşina pe care lucraţi, folosiţi comanda:

```
cat /etc/services | grep rpc
```

Atunci când un program client doreşte să apeleze un anumit serviciu RPC, el nu ştie pe ce port rulează acesta pentru a face apelul. Trebuie să existe o metoda de aflare a acestui port. Deamon-ul portmapper rezolva aceasta problema. Atunci când un client face un apel RPC, mai întâi apelează serviciul de portmapper pentru aflarea portului serverului.

Pornirea/oprirea şi alte operaţii asupra portmapper-ul se face ca în exemplul de mai jos (înainte de a apela **`/etc/init.d/portmap`** verificaţi că locaţia pentru **`portmap`** este cea bună). Portmapper-ul trebuie pornit înaintea oricărui server RPC. Dacă portmapper-ul este repornit trebuie repornite şi servere-le RPC.

```
# /etc/init.d/portmap
Usage: /etc/init.d/portmap {start|stop|restart|force-reload|reload|status}

# /etc/init.d/portmap start
Starting RPC portmap daemon                                          done

# /etc/init.d/portmap status
Checking for RPC portmap daemon:                                     running
```

Pentru a verifica maparea dintre protocolul RPC si protocolul de nivel transport se poate folosi comanda de mai jos:

```
# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
```

O procedura la distanţă oferită de un server este identificata în mod unic prin număr program, număr versiune, număr procedură. **Numărul de program** identifica un număr de proceduri înrudite, fiecare din aceste având un **număr unic de procedură**.

Un program poate avea mai multe versiuni. Fiecare versiune are un număr de proceduri care pot fi apelate la distanta. Numărul de versiune permite posibilitatea existenţei simultane a mai multor versiuni ale unui protocol RPC. Fişierul */etc/rpc* mapează numele de serviciu cu numărul de program dat:

```
# cat /etc/rpc
#ident  "@(#)rpc        1.11    95/07/14 SMI"   /* SVr4.0 1.2   */
#
#       rpc
#
portmapper      100000  portmap sunrpc rpcbind
rstatd          100001  rstat rup perfmeter rstat_svc
rusersd         100002  rusers
nfs             100003  nfsprog
ypserv          100004  ypprog
mountd          100005  mount showmount
ypbind          100007
walld           100008  rwall shutdown
yppasswdd       100009  yppasswd
etherstatd      100010  etherstat
rquotad         100011  rquotaprog quota rquota
sprayd          100012  spray
...
```

2.1.2. RPC in Windows

În Windows serviciul de RPC poarta numele RpcSs şi este pornit de către serviciul Svchost.exe, care este un proces gazdă generic pentru serviciile care se execută pornind de la biblioteci cu legare dinamică (DLL-uri). La pornire, Svchost.exe verifică serviciile care sunt parte a registry pentru a construi o listă a serviciilor pe care trebuie să le încarce. Fiecare sesiune Svchost.exe poate conţine o grupare de servicii. În consecinţă, se pot executa servicii distincte, în funcţie de modul şi de locul în care s-a lansat Svchost.exe. Această grupare a serviciilor permite un control mai bun şi o depanare mai simplă.

RpcSs este pornit pe portul 135. Pentru a vedea care sunt parametri folosiţi de RPC verificaţi:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\RpcSs
```

Pentru a vedea o listă a serviciilor care se execută în Svchost şi pentru a verifica că serviciul de RPC este pornit executaţi în linia de comandă **`tasklist /SVC`**:

```
> tasklist /SVC
    Image Name          PID      Services
   =========================================================================
   System Process         0     N/A
   System                 8     N/A
   Smss.exe             132     N/A
   Csrss.exe            160     N/A
   Winlogon.exe         180     N/A
   Services.exe         208     AppMgmt,Browser,Dhcp,Dmserver,Dnscache,
                                Eventlog,LanmanServer,LanmanWorkstation,
                                LmHosts,Messenger,PlugPlay,ProtectedStorage,
                                Seclogon,TrkWks,W32Time,Wmi
   Lsass.exe            220     Netlogon,PolicyAgent,SamSs
   Svchost.exe          404     RpcSs
   Spoolsv.exe          452     Spooler
   Cisvc.exe            544     Cisvc
   Svchost.exe          556     EventSystem,Netman,NtmsSvc,RasMan,
                                SENS,TapiSrv
   Regsvc.exe           580     RemoteRegistry
   Mstask.exe           596     Schedule
   Snmp.exe             660     SNMP
   Winmgmt.exe          728     WinMgmt
   Explorer.exe         812     N/A
   Cmd.exe             1300     N/A
   Tasklist.exe        1144     N/A
```

Service Name	**RpcSs**	Process Name	**svchost -k rpcss**
Default Settings	**XP Home**: Manual	**XP Pro**: Manual	
		XP Pro w/SP2: Automatic	
Microsoft Service Description	Provides the endpoint mapper and other miscellaneous RPC services.		
Dependencies	-	-	
Real World Description	Nobody is exactly sure what this service does, but kill it off and watch your system quickly die. It's a fact that a multitude of the other services depend on this service running.		
Is this service needed?	**Yes !**	Recommended Setting:	**Automatic**
Note	Updated to reflect SP2 changes.		

Figura 2.1: Descrierea RpcSs

Pentru a obţine informaţii suplimentare despre un serviciu pornit se poate folosi comanda: **`tasklist /FI "PID eq IDproces"`**. De exemplu, pentru RPC pe sistemul care a fost dat exemplu de mai sus se poate folosi comanda **`tasklist /FI "PID eq 404"`**.

Descrierea serviciului RpcSs oferită de Microsoft este dată în Figura 2.1. Majoritatea aplicaţiilor şi serviciilor Windows sunt bazate pe serviciul de RPC. Aşa cum se poate observa din Figura 2.2, serviciul de RPC nu poate fi oprit.

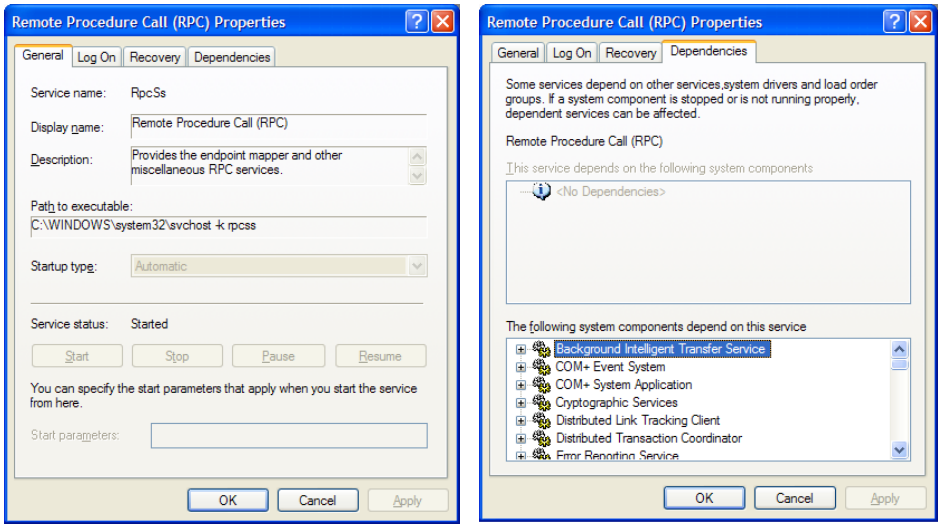


Figura 2.2: Caracteristici RPC pe Windows. Servicii dependente de RpcSs

Aşa cum se poate observa, serviciul de RPC nu poate fi oprit deoarece sunt multe servicii care depinde de el şi care nu ar mai funcţiona. Dezvoltarea unei aplicaţii RPC pe Windows va fi tratată în secţiunea următoare.

2.2. Rpcgen

`Rpcgen` poate genera automat codului pe baza specificaţiilor din interfaţa de definirea a serviciului de RPC. Fişierul de intrare pentru `rpcgen` este un program remote de specificaţie a interfeţei care practic specifică protocolul deoarece structurile de date transmise afectează protocolul între partea client şi server. Prin convenţie fişierul de specificare a interfeţei are extensia `.x`. Programul `rpcgen` foloseşte numele fişierului pentru a crea patru fişiere de ieşire. Pentru a finaliza aplicaţia programatorul trebuie să scrie procedura principală pentru partea client şi procedurile remote pentru partea server. `Rpcgen` este disponibil pe majoritatea sistemelor UNIX. Cuvintele cheie (definiţiile) folosite de rpcgen sunt: `constant`, `enumeration`, `struct`, `union`, `typedef`, `program`.

Pentru dezvoltarea unei aplicaţii RPC trebuie urmaţi paşii :

1. Construirea unui protocol pentru comunicaţia client/server.
2. Dezvoltarea unui program server.
3. Dezvoltarea unui program client.

În această secţiune vom trata primul pas, urmând ca ceilalţi doi paţi să fie parcurşi în următoarele două secţiuni.

EXEMPLUL 1. Pentru exemplificare vom trata în cele ce urmează un exemplu de aplicaţie client-server bazat pe RPC. Se doreşte scrierea unei aplicaţii care afişează încărcarea unui server (Load-ul).

Programul de afişare a încărcării are o funcţie de aflare a valorii acestui parametru:

```
double  get_load (void);
```

Programul ce defineşte această funcţie ca procedură remote (funcţia `get_load`) este fişierul de specificaţie `load.x`. Programul are o un identificator, o versiune. Fiecare funcţie definită are un număr de versiune.

```
program LOAD_PROG {
        version LOAD_VERS {
                double GET_LOAD(void) = 1;
        } = 1;
} = 123456789;
```

Se poate observa că fiecare dintre funcţii are un singur argument. Funcţiile definite într-o astfel de interfaţă vor avea maxim un parametru `in` şi un parametru `out`. Dacă sunt necesari mai mulţi parametri, aceştia trebuie reprezentaţi printr-o structură. Prin convenţie numele programului, versiunea şi numele procedurii sunt scrise cu majuscule. Fiecare funcţie trebuie definită şi este numerotată (începând cu valoarea 1). Specificaţia defineşte un program care va fi executat remote şi va fi format din aceste funcţii. Programul are un nume, un număr de versiune şi un număr unic de identificare (ales de programator).

Cuvinte cheie sunt `program` şi `version`. `Rpcgen` creează numele procedurilor prin translatarea numelui la minuscule şi prefixarea lor de '_' urmat de numărul versiunii.

Numele filtrelor pentru codificarea şi decodificarea datelor rezultă din concatenarea şirului "`xdr_`" cu numele datei. Tipurile de date acceptate de XDR includ: `int`, `unsigned int`, `long`, `structure`, şir de lungime fixă, `string` şi date codificate binar.

Compilarea specificaţiilor se face prin intermediul comenzii `rpcgen`, utilizând diverşi parametri, printre care şi numele fişierului `.x`.

Rezultatele compilări şi folosirea lor pentru generarea programelor client şi server vor fi prezentate în secţiunea 2.5.

Studiul de caz va acoperi şi situaţia folosirii de structuri în cadrul specificaţiilor `rpcgen`.

2.3. Implementarea unui Server RPC

Operaţiile de acces ale unei proceduri la distanţă sunt descrise în Figura 2.3. Acestea sunt:

1. Clientul apelează pentru procedura respectivă un ”stub” local. Aceasta împachetează parametrii (marshalling) într-un mesaj ce va fi transmis prin reţea.
2. Funcţiile de reţea din nucleul SO sunt apelate de stub pentru a transmite mesajul prin reţea.
3. Nucleul transmite mesajul (mesajele) la host-ul remote folosind un transport cu sau fără conexiune.
4. Stub-ul server despachetează argumentele din mesajul recepţionat.
5. Stub-ul server execută apelul de procedură locală.
6. Terminarea execuţiei procedurii şi revenirea controlului la stub-ul server.
7. Stub-ul server împachetează rezultatul într-un mesaj răspuns.
8. Mesajul de răspuns este transmis prin reţea.
9. Stub-ul client citeşte mesajul folosind funcţiile de reţea.
10. Rezultatul este despachetat şi valorile întoarse sunt depuse pe stivă pentru procesul local.

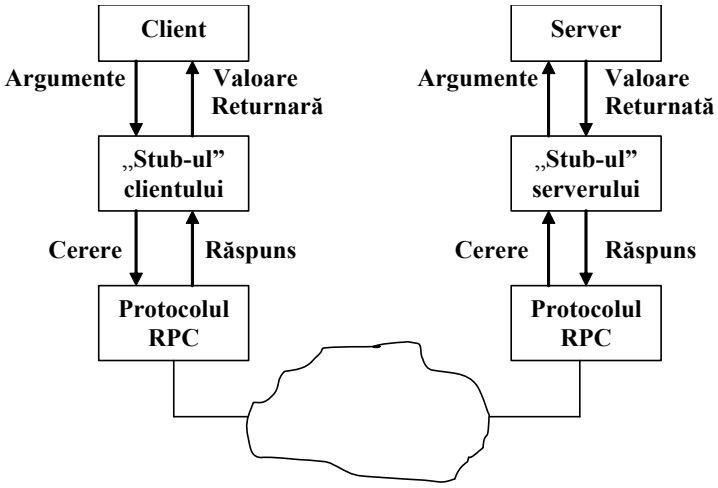


Figura 2.3. Mecanismul RPC

Comunicarea în reţea folosind socket-uri se bazează pe transmisia mesajelor de la un proces la altul. Aşa cum am văzut, apelul unei proceduri remote creează iluzia unui apel de procedură obişnuit, astfel încât comunicarea prin reţea să fie transparentă utilizatorului. Procesul execută un apel de procedură remote prin

transferul parametrilor de apel şi adresei de revenire pe stivă şi efectuează un salt la codul de început al procedurii. Procedura în sine este cea care se ocupă de comunicarea prin reţea. După terminarea execuţiei procedurii remote controlul revine la procesul apelant, care îşi continuă execuţia.

RPC-ul ar trebui să asigure transparenţa comunicării prin reţea, aşa încât totul să arate ca un apel de procedură uzual.

Codul programului server este prezentat mai jos. Aşa cum vom vedea în secţiunea care tratează compilarea aplicaţiilor, vom vedea că server-ul trebuie să implementeze funcţii cu numele specificate în `load.x` precedate de "`_1_svc`".

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

#include "load.h"

double * get_load_1_svc(void *p, struct svc_req *cl){

        static double load;

       FILE *fp;
       fp = popen("uptime | cut -d ',' -f4 | sed 's/^ //'","r");

       fscanf(fp, "%lf", &load);

       fclose(fp);
       return &load;
}
```

Se observă în funcţia pe care server-ul o foloseşte avem declararea variabilei `load` de clasă `static` care garantează că este returnată o adresă validă.

De asemenea se observă lipsa funcţiei `main`. Aceasta este prezentă în stub-ul generat automat. Al doilea parametru al funcţiilor implementate de server este un pointer la o structură de tipul `struct svc_req`. Această structură conţine informaţii cu privire la context în timpul unei invocări: programul, versiunea, numerele de procedură, credenţialele şi un pointer la o structură `SVCXPRT` care conţine informaţii de transport.

```
/* Service request */
struct svc_req {
         u_int32_t rq_prog;           /* service program number    */
         u_int32_t rq_vers;           /* service protocol version  */
         u_int32_t rq_proc;           /* the desired procedure     */
         struct opaque_auth rq_cred;  /* raw creds from the wire   */
         caddr_t rq_clntcred;         /* read only cooked cred     */
         SVCXPRT *rq_xprt;            /* associated transport      */
};
```

2.4. Implementarea unui Client RPC

Pentru implementarea clientului, apelul de funcţie remote se face cu ajutorul pointer-ului către conexiunea creată. Programul client implementează funcţia `main` în care face apelul de procedură.

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

#include "load.h"
#define RMACHINE "localhost"

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

        /* variabila clientului */
        CLIENT *handle;

        double *res;

        handle=clnt_create(
                RMACHINE,       /* numele masinii unde se afla server-ul */
                LOAD_PROG,      /* numele programului disponibil pe server */
                LOAD_VERS,      /* versiunea programului */
                "tcp");         /* tipul conexiunii client-server */

        if(handle == NULL) {
                perror("");
                return -1;
        }
        res = get_load_1(NULL, handle);
        printf( "The server load is:  %lf\n", *res);

        return 0;
}
```

Apelul funcţiei `clnt_create` are ca prim parametru numele maşinii care găzduieşte server-ul. În exemplul dat am considerat că server-ul şi clientul rulează pe aceeaşi maşină (`#define RMACHINE "localhost"`). Putem transmite numele maşinii unde se află scerver-ul prin parametrii liniei de comandă. Observaţi că nu trebuie specificat portul pentru stabilirea conexiunii, descoperirea lui fiind făcută de portmapper.

2.5. Compilarea aplicaţiilor RPC pe sisteme Linux/Solaris

Considerând exemplul tratat în secţiunile anterioare, să presupunem că aplicaţia client se numeşte `rpc_client.c` iar aplicaţia server se numeşte `rpc_server.c` şi protocolul a fost definit în `load.x`.

Execuţia `rpcgen` generează trei (sau patru) fişiere:

- `load.h`: fişierul antet folosit de programul client şi server.
- `load_clnt.c` - sursa C ce conţine setul de funcţii stub pentru client.
- `load _svc.c` - sursa C ce conţine setul de funcţii stub utilizate de server.
- `load _xdr.c` - apeluri de proceduri XDR folosite în client şi server pentru transferul argumentelor (dacă au fost definite structuri de date în `load.x`).

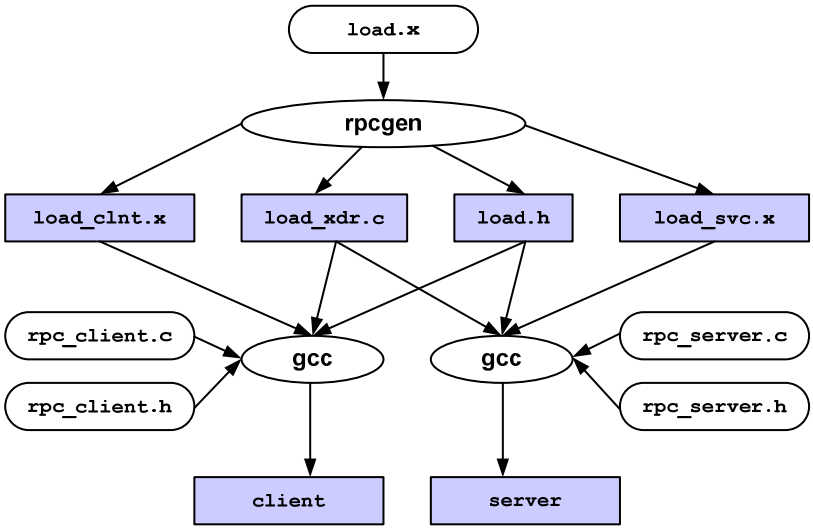


Figura 2.4. Folosirea rpcgen în dezvoltarea aplicaţiilor distribuite.

Funcţiile sunt generate din specificaţie astfel:

- numele funcţiilor este prefixat de "`_1`", numele funcţiei fiind scris cu minuscule.
- la client: funcţiile generate au doi parametri, iar la server conform specificaţiei.
- la client: al doilea parametru este un "`handle`" creat de apelul `clnt_create`.
- la client/server: valoarea returnată de funcţie este înlocuită de un pointer la acea valoare.

Programele client şi server trebuie să includă fişierul de definiţii `load.h` (`#include load.h`). Pentru exemplul considerat, deoarece nu s-au folosit structuri, nu se generează `load_xdr.c`.

Fişierul `makefile` necesar pentru compilarea aplicaţiei este:

```
build:
	rpcgen -C load.x
	gcc -o server rpc_server.c load_svc.c -lnsl -Wall
	gcc -o client rpc_client.c load_clnt.c -lnsl -Wall

clean:
	rm -f client server load.h load_svc.c load_clnt.c
```

Fişierul `load.h` generat este:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _LOAD_H_RPCGEN
#define _LOAD_H_RPCGEN

#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

#define LOAD_PROG 123456789
#define LOAD_VERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define GET_LOAD 1
extern  double * get_load_1(void *, CLIENT *);
extern  double * get_load_1_svc(void *, struct svc_req *);
extern int load_prog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define GET_LOAD 1
extern  double * get_load_1();
extern  double * get_load_1_svc();
extern int load_prog_1_freeresult ();
#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_LOAD_H_RPCGEN */
```

Fişierul `load_clnt.c` generat este:

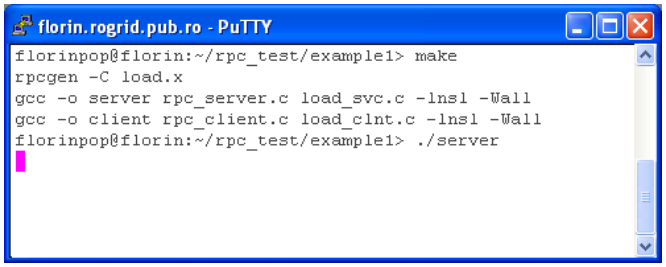
```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "load.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

double * get_load_1(void *argp, CLIENT *clnt){
        static double clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, GET_LOAD,
                (xdrproc_t) xdr_void, (caddr_t) argp,
                (xdrproc_t) xdr_double, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

Fişierul `load_svc.c` generat este:

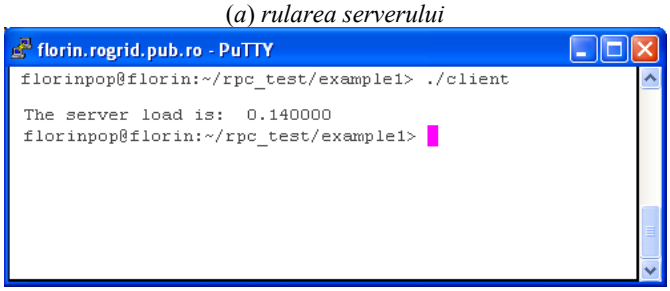
```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "load.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void load_prog_1(struct svc_req *rqstp, register SVCXPRT *transp){
   union {
        int fill;
   } 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 GET_LOAD:
            _xdr_argument = (xdrproc_t) xdr_void;
            _xdr_result = (xdrproc_t) xdr_double;
            local = (char *(*)(char *, struct svc_req *)) get_load_1_svc;
            break;
        default:
            svcerr_noproc (transp);
            return;
   }

   memset ((char *)&argument, 0, sizeof (argument));
   if (!svc_getargs(transp, (xdrproc_t)_xdr_argument, (caddr_t)&argument)) {
        svcerr_decode (transp);
        return;
   }

   result = (*local)((char *)&argument, rqstp);
   if(result!=NULL && !svc_sendreply(transp,(xdrproc_t)_xdr_result,result)){
        svcerr_systemerr (transp);
   }

   if(!svc_freeargs (transp, (xdrproc_t)_xdr_argument, (caddr_t)&argument)){
        fprintf (stderr, "%s", "unable to free arguments");
        exit (1);
   }
   return;
}

int main (int argc, char **argv){
   register SVCXPRT *transp;
```

```
   pmap_unset (LOAD_PROG, LOAD_VERS);

   transp = svcudp_create(RPC_ANYSOCK);
   if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
   }
   if(!svc_register(transp,LOAD_PROG,LOAD_VERS,load_prog_1, IPPROTO_UDP)) {
     fprintf(stderr, "%s","unable to register (LOAD_PROG,LOAD_VERS, udp).");
     exit(1);
   }

   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create tcp service.");
        exit(1);
   }
   if(!svc_register(transp,LOAD_PROG,LOAD_VERS, load_prog_1, IPPROTO_TCP)) {
     fprintf(stderr,"%s","unable to register (LOAD_PROG,LOAD_VERS, tcp).");
     exit(1);
   }

   svc_run ();
   fprintf (stderr, "%s", "svc_run returned");
   exit (1);
   /* NOTREACHED */
}
```

La final se vor genera doua fişiere binare ce încapsulează aplicaţiile client şi server. Execuţia lor este prezentată în Figura 2.5.



(*a*) *rularea serverului*



(*b*) *rularea clientului*

Figura 2.4. Execuţia clientului şi serverului

Tipurile de erori ce pot apărea şi trebuie tratate la folosirea RPC şi soluţiile aferente sunt:

- *Server-ul nu poate fi localizat*: trebuie returnat un cod de eroare.
- *Cererea spre server s-a pierdut*: client-ul poate poziţiona un timer care expiră dacă răspunsul nu soseşte în timpul aşteptat. Dacă server-ul este lent s-ar putea considera că o cerere s-a pierdut, ceea ce este greşit. Pentru a evita execuţia unei cereri de două sau mai multe ori se poate include în mesaj un identificator.
- *Răspuns de la server pierdut*: eroarea se tratează similar celei anterioare.
- *Server căzut*: verificarea handler-ului clientului.
- *Client căzut*: poate fi ţinută evidenţa mesajelor RPC transmise pe disc.

În ciuda eleganţei conceptului de apel al procedurii la distanţă, implementarea generează o serie de probleme. Una dintre ele este folosirea pointerilor. În mod normal un pointer la o procedura nu este o problema. Procedura apelată poate folosi pointerii în acelaşi mod în care ii poate folosi şi apelantul pentru ca cele două proceduri sunt localizate in acelaşi spaţiu de adrese virtuale. Pasarea pointerilor în cazul apelului procedurilor la distanta este imposibil pentru ca clientul şi serverul rezida în spatii de adrese diferite.

Există situaţii în care se poate folosi totuşi pasarea pointerilor. Să presupunem că primul parametru este un pointer la un întreg, k. Stub-ul clientului poate împacheta întregul şi îl poate trimite în acest fel spre server. După aceea stub-ul serverului creează un pointer la întregul k pe care îl pasează procedurii serverului. Atunci când procedura serverului cedează controlul stub-ului serverului acesta trimite înapoi la client întregul, unde vechea valoare a întregului este suprascrisa cu noua valoare a întregului. De fapt, secvenţa standard de apelare prin referinţă a fost înlocuită prin copiere şi reactualizare (copy-restore). Din păcate această metodă nu funcţionează în toate cazurile. De exemplu dacă pointerul se referă la o structură complexă nu mai este posibilă aceasta metoda. Din aceste considerente trebuie făcute anumite restricţii pentru parametrii pasaţi în apelul procedurilor la distantă.

O altă problema este constituită de posibilitatea declarării unui vector fără să fie nevoie să se specifice dimensiunea maximă a acestuia. Fiecare vector are o dimensiune maximă cunoscuta numai de procedura apelantă şi procedura apelată. Într-o astfel de situaţie împachetarea de către stub-ul clientului a parametrilor este imposibilă pentru că acesta nu poate determina dimensiunile maxime ale vectorilor.

O a treia problema legată de deducerea tipului parametrilor. Un exemplu este funcţia printf, care poate avea oricâţi parametrii de tipuri diferite: întregi, caractere, stringuri, etc. Încercarea de a apela la distanta funcţia printf este practic imposibilă pentru că limbajul C este prea îngăduitor.

Altă problema se referă la utilizarea variabilelor globale. În mod obişnuit, apelarea si procedura apelantă şi procedura apelată pot comunica folosind variabile globale ca parametrii. Dacă procedura apelată este mutată pe o maşină la distanţă codul nu va putea fi executat pentru că variabilele globale nu mai sunt partajate.

2.6. XML-RPC

Un dintre limitările RPC este aceea că nu acesta este suficient pentru crearea de aplicaţii distribuite orientate obiect, pentru care comunicaţia între procesele aflate în spaţii de adresare diferite se face prin obiecte.

O tehnica pentru rezolvarea acestei probleme este Java RMI (Remote Method Invocation). Dar Java RMI a fost proiectat numai pentru limbajul Java. Astfel, toate componentele sistemului distribuit dorit trebuie scrise în Java pentru a funcţiona corect (http://www.ecst.csuchico.edu/~amk/foo/advjava/notes/rminotes.html).

O altă soluţie este JERI (Jini Extensible Remote Invocation). Acesta este o extindere a RMI-ului, introdusa in Jini 2.0. Faţă de RMI, JERI are adăugat mecanismul de securitate puternic şi mult mai dezvoltat şi uşor configurabil. (http://www.javaworld.com/javaworld/jw-12-2003/jw-1219-jiniology_p.html).

Web services sunt un set de utilitare care permit construirea de aplicaţii distribuite deasupra infrastructurii web existente. Aceste aplicaţii utilizează web-ul ca un fel de „nivel transport".

XML-RPC este printre cele mai uşoare metode de construire a serviciilor de web si permite apelul uşor al procedurilor la distanta. XML (eXtensible Markup Language) furnizează un „vocabular" pentru descrierea procedurilor la distanta (RPC) care sunt apoi apelate folosindu-se HTTP (HyperText Transfer Protocol).

Un apel XML-RPC este condus intre doua parţi: un client (procesul apelant) şi un server (procesul apelat). Serverul este făcut disponibil la un anumit URL (de exemplu http://se.rogrid.pub.ro/rpcserv). Pentru a utiliza procedurile de la acest server sunt necesari următorii paşi:

1. Programul client face un apel la distanta utilizând un client XML-RPC in care specifica numele metodei apelate, parametrii si serverul ţinta.
2. Clientul XML-RPC ia numele metodei si parametrii acesteia si ii împachetează ca mesaj XML. Apoi clientul face un apel HTTP POST care conţine informaţiile specificate XML cate serverul ţinta.
3. Serverul HTTP de pe maşina ţinta primeşte cererea POST si pasează conţinutul XML la serverul XML-RPC care asculta.
4. Serverul XML-RPC parsează mesajul XML primit pentru a găsi numele metodei si parametrii acesteia iar apoi apelează metoda specificata.
5. Metoda apelata returnează un răspuns la procesul XML-RPC care începe sa-l împacheteze ca un răspuns XML.
6. Serverul de web returnează acest răspuns XML la cererea HTTP POST.
7. Clientul XML-RPC parsează răspunsul XML pentru a găsi rezultatul apelului, pe care-l întoarce programului client.
8. Programul client îşi continua execuţia.

Un proces poate fi atât server (pentru unii clienţi) cat si client (pentru alte servere). Utilizarea protocolului HTTP face ca apelurile XML_RPC sa fie sincrone şi stateless.

Un apel XML-RPC este urmat de exact un răspuns, acesta fiind sincron cu cererea. Acest lucru se întâmpla pentru ca răspunsul trebuie să se producă pe aceeaşi conexiune HTTP ca si cererea.

HTTP este un protocol fără stare (stateless). Aceasta înseamnă că nici un context nu este reţinut de la o cerere la următoarea. XML-RPC moşteneşte aceasta caracteristica. Deci, daca un program client invoca de mai multe ori aceeaşi metoda de pe server, XML-RPC tratează aceste cereri total separat. Acest lucru ocoleşte overhead-ul implicat in menţinerea stării pe mai multe sisteme.

Cererea în XML:

```
POST /rpchandler HTTP/1.0
User-Agent: AcmeXMLRPC/1.0
Host: xmlrpc.example.com
Content-Type: text/xml
Content-Length: 165

<?xml version="1.0"?>

<methodCall>
        <methodName>getCapitalCity</methodName>
        <params>
                <param>
                        <value><string>England</string></value>
                </param>
        </params>
</methodCall>
```

Răspunsul în XML:

```
<?xml version="1.0"?>
<methodResponse>
        <params>
                <param>
                        <value><string>London</string></value>
                </param>
        </params>
</methodResponse>
```

Protocoalele alternative XML-RPC pentru creare de Web Services sunt :

- SOAP (Simple Object Access Protocol): http://webservices.xml.com/pub/a/ws/2000/11/01/protocols/quickref.html#soap;
- UDDI (Universal Description, Discovery, and Integration): http://webservices.xml.com/pub/a/ws/2000/11/01/protocols/quickref.html#uddi;
- WSDL (Web Services Description Language).

2.7. Studiu de caz: Monitorizarea servere-lor folosind RPC

Monitorizarea unor sisteme de calcul presupune colectarea unor parametri care pot descrie starea acestora (utilizarea procesorului, încărcarea, memoria disponibilă, spaţiul liber pe disc etc.) şi prezentarea acestor parametri într-un mod uşor de urmărit şi interpretat.

EXEMPLUL 2. Vom implementa un serviciu de monitorizare pentru sistemele de calcul din cadrul unei universităţi. Acest serviciu va fi o aplicaţie client – server cu un model de comunicaţie/colectare de parametri bazat pe RPC.

Arhitectura serviciului este prezentată în Figura 2.5. Serviciul este format din trei entităţi:

- *Serverul studentului*. Această componentă are două funcţionalităţi: configurarea clienţilor student pentru obţinerea parametrilor de monitorizare şi afişarea rezultatelor în format HTML.
- *Clientul studentului*. La pornire, clientul studentului face o cerere RPC către serverul local pentru a cere informaţiile de configurare (lista parametrilor de monitorizare şi intervalul de timp între două cereri succesive). După operaţia de configurare, clientul va face cereri serverului facultăţii pentru obţinerea parametrilor de monitorizare (cererile de fac la intervale de timp constante).
- *Serverul facultăţii*. Acest server are ca rol obţinerea parametrilor de monitorizare pentru sistemul de calcul al facultăţii în funcţie de cererile clienţilor studenţilor şi oferirea rezultatelor.

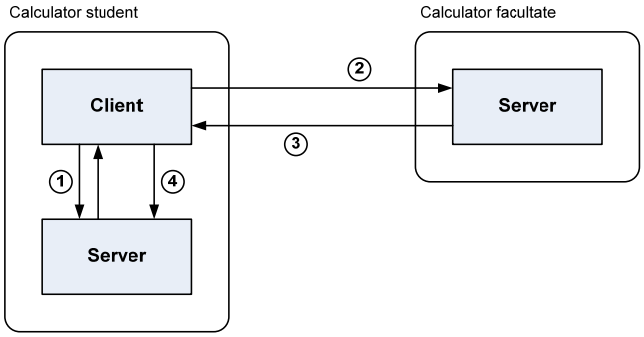


Figura 2.5. Arhitectura serviciului de monitorizare

Acţiunile executate de sistemul de monitorizare sunt:
(1) Cerere / răspuns RPC de configurare între clientul şi serverul studentului.
(2) Cerere RPC pentru obţinerea parametrilor de monitorizare din partea clientului studentului pentru serverul facultăţii.
(3) Răspuns RPC din partea server-ului facultăţii cu parametri de monitorizare.

(4) Clientul student trimite (RPC) rezultatele monitorizării către serverul local care va scrie rezultatele într-un format HTML.

Serverul studentului şi serverul facultăţii vor si implementate ca un singur server. Acest lucru ar putea ajuta la compunerea serviciilor de monitorizare. Serverul studentului va fi rulat cu comanda:

```
# ./server
```

Acesta, la o cerere de configurare din partea unui client, va citi fişierul de configurare (`config.txt`) cu structura:

```
MonitoringInterval = 180
Parameters = Load5 CPUFreq MemSwap MemFree DiskFree OSType OSVersion
Hostname
```

unde `MonitoringInterval` este exprimat în secunde. Lista de parametri (separaţi printr-un spaţiu) poate conţine toţi parametri (ca în exemplul dat) sau mai puţini parametri. Rezultatul monitorizării primit de la client se va afişa sub formă de tabel. Fişierul de ieşire `result.html` are următoarea structură:

```
Hostname: <b>e420</b>
<table>
 <tr> <td style='background:#E6E6E6'> OSType    </td>
      <td style='background:#E6E6E6'> sun4u     </td> </tr>
 <tr> <td style='background:#E6E6E6'> OSVersion </td>
      <td style='background:#E6E6E6'> 5.9       </td> </tr>
 <tr> <td style='background:#CCFFFF'> CPUFreq(1)</td>
      <td style='background:#CCFFFF'> 450MHz    </td> </tr>
 <tr> <td style='background:#CCFFFF'> CPUFreq(2)</td>
      <td style='background:#CCFFFF'> 450MHz    </td> </tr>
 <tr> <td style='background:#FFFF99'> Load5     </td>
      <td style='background:#FFFF99'> 0.09      </td> </tr>
 <tr> <td style='background:#FF99CC'> MemSwap   </td>
      <td style='background:#FF99CC'> 4142,97MB </td> </tr>
 <tr> <td style='background:#FF99CC'> MemFree   </td>
      <td style='background:#FF99CC'> 606,12MB  </td> </tr>
 <tr> <td style='background:#CCFFCC'> DiskFree  </td>
      <td style='background:#CCFFCC'> 92931,19MB</td> </tr>
</table>
```

Hostname: **e420**

OSType	sun4u
OSVersion	5.9
CPUFreq(1)	450MHz
CPUFreq(2)	450MHz
Load5	0.09
MemSwap	4142,97MB
MemFree	606,12MB
DiskFree	92931,19MB

Serverul facultăţii va fi rulat cu comanda:

```
# ./server
```

Acesta, la primirea unei cereri de monitorizare din partea unui client, va trebui să întoarcă valorile parametrilor solicitaţi. Pentru a obţine aceşti parametri serverul va apela o serie de comenzi de sistem şi va procesa răspunsul acestora.

În continuare sunt prezentate câteva comenzi de sistem din care au fost obţinute datele din tabelul prezentat ca rezultat al monitorizării.

- Comanda `uname –a` pentru Linux

```
# uname -a
Linux florin 2.6.18.2-34-xen #1 SMP Mon Nov 27 11:46:27 UTC 2006 x86_64
x86_64 x86_64 GNU/Linux
```

- Comanda `uname –a` pentru Solaris

```
# uname -a
SunOS e420 5.9 Generic_117171-02 sun4u sparc SUNW,Ultra-80
```

- Comanda `uptime`, identica pentru ambele sisteme de operare

```
# uptime
12:24pm  up 11 day(s), 13:29,  1 user,  load average: 0.18, 0.09, 0.21
```

- Comanda `prtdiag` pentru Solaris

```
# prtdiag
System Configuration:   Sun Microsystems   sun4u Sun Enterprise 420R (2 X
UltraSPARC-II 450MHz)
System clock frequency: 113 MHz
Memory size: 1024 Megabytes

========================= CPUs =========================
                     Run   Ecache   CPU    CPU
Brd  CPU   Module    MHz     MB    Impl.   Mask
---  ---  -------  -----  ------  ------  ----
 0     1      1      450     4.0   US-II    10.0
 0     2      2      450     4.0   US-II    10.0

======================== IO Cards =========================
     Bus   Freq
Brd  Type  MHz   Slot        Name                          Model
---  ----  ----  ----------  ----------------------------  -----------------
 0   PCI    33     On-Board  network-SUNW,hme
 0   PCI    33     On-Board  scsi-glm/disk (block)         Symbios,53C875
 0   PCI    33     On-Board  scsi-glm/disk (block)         Symbios,53C875
 0   PCI    33         pcib  SUNW,hme-pci108e,1001        SUNW,qsi-cheerio
 0   PCI    33  pcia slot 1  TSI,gfxp                      GFXP

No failures found in System
===========================
```

- Conţinutul fişierului `/proc/cpuinfo` (pentru Linux):

```
# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 4
model name      : Intel(R) Pentium(R) 4 CPU 3.20GHz
stepping        : 3
```

```
cpu MHz         : 3212.220
cache size      : 2048 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu tsc msr pae mce cx8 apic mtrr mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm syscall nx lm constant_tsc pni
monitor ds_cpl est cid cx16 xtpr
bogomips        : 8033.61
clflush size    : 64
cache_alignment : 128
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

- Comanda `df -k`, pentru Linux

```
# df -k
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             41944376   7290584  34653792  18% /
udev                   1220528       136   1220392   1% /dev
/dev/mapper/VolGroup00-VolLog01
                      126456040  63165996  63290044  50% /home
/dev/mapper/VolGroup00-VolLog00
                       42162932   2475460  39687472   6% /opt
/dev/sda1              80324968  10089028  70235940  13% /windows
```

- Comanda `df -k`, pentru Solaris

```
# df -k
Filesystem             kbytes    used   avail capacity  Mounted on
/dev/md/dsk/d0       13246819 12893123  221228  99%    /
/proc                       0       0       0     0%    /proc
mnttab                      0       0       0     0%    /etc/mnttab
fd                          0       0       0     0%    /dev/fd
/dev/md/dsk/d3       2056211  662864 1331661    34%    /var
swap                 4221320      40 4221280     1%    /var/run
swap                 4221320      40 4221280     1%    /tmp
hpcdom:/raid5-2      69506363 53795924 15015376 79%    /export
hpcdom:/raid5-1      69506363 18595898 50215402 28%    /raid
hpcdom:/raid0-1/bsm/e420
                     69506363 40433431 28377869 59%    /etc/audit/e4202.
```

- Comanda `vmstat`, pentru Linux

```
# vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 0  0      0 373548 111768 1357196    0    0    46    17   71  110  0  0 98  1
```

- Comanda `vmstat`, pentru Solaris

```
# vmstat
kthr      memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr m0 m1 m3 m1   in   sy   cs us sy id
0 0 0 4242400 620664 28 127 4  5  5  0  0  3  0  1  2  551 1117  317  1  2 97
```

Clientul studentului va porni cu comanda:

```
# ./client [server_ip|server_host]
```

unde `server_ip|server_host` specifică locaţia serverului facultăţii.

Acesta va face o cerere de configurare la server-ul local. După obţinerea informaţiilor va face cereri către serverul facultăţii pentru a obţine parametri de monitorizare. La primirea răspunsului de la serverul facultăţii, clientul va trimite datele serverului local pentru afişare.

Protocolul de comunicaţie client-server se va construi pe baza protocolului RPC. Clientul şi serverul se vor implementa în limbajul C. Protocolul de comunicaţie se va compila cu *rpcgen*. Iată un exemplu de `makefile` pentru compilarea aplicaţiei:

```
compile: client server

client: mon_clnt.c client.c mon_xdr.c
	cc -lnsl $^ -o $@

server: mon_svc.c server.c mon_xdr.c
	cc -lnsl $^ -o $@

mon_clnt.c mon_svc.c mon_xdr.c mon.h: mon.x
	rpcgen -C mon.x

clean:
	rm -f client server mon_svc.c mon_clnt.c mon_xdr.c mon.h result.html
```

Problema reprezentării parametrilor procedurii este foarte importantă. Dacă o procedură are, spre exemplu, trei parametrii de tipurile întreg scurt, string şi întreg, se pune problema împachetării lor pentru transmisie astfel încât la despachetarea lor să se obţină parametrii iniţiali. De exemplu, întregul scurt poate fi reprezentat pe primii doi octeţi şi lăsând liberi următorii doi sau invers. String-ul poate fi prefixat de lungimea sa sau terminat cu un simbol special. Întregul poate fi reprezentat ca “big-endian” sau “little-endian”.

Fişierul `mon.x` pentru descrierea protocolului este:

```
#define MAX_HOST_NAME   1024
#define MAX_OS_TYPE      128
#define MAX_OS_VERSION    32
#define MAX_PROCS      65536

struct Data {
	int     result;
	int     whatToUse;
	string  hostName  <MAX_HOST_NAME>;
	string  osType    <MAX_OS_TYPE>;
	string  osVersion <MAX_OS_VERSION>;
	float   cpuFreq   <MAX_PROCS>;
	float   load5;
	float   memSwap;
	float   memFree;
	double  diskFree;
};

struct ConfigData {
	int     result;
	int     monitoringInterval;
```

```
        int     whatToUse;
};

program MON {

   version MONVERS {

       ConfigData READ_CONFIG_DATA (void) = 1;
       Data       READ_DATA        (int)  = 2;
       int        WRITE_DATA       (Data) = 3;

   } = 1;

} = 0x23251249;
```

Pentru a trata unitar reprezentările datelor RPC de sub Sun utilizează un format standard numit XDR. Ordinea este "big-endian" şi dimensiunea minimă a fiecărui câmp este fixată la 32 de biţi. Mesajul poate fi format folosind tipuri implicite. Aşadar sunt trimise numai valori şi se presupune că atât clientul cât şi server-ul cunosc tipul datelor cu care lucrează. Un pointer referă o adresă din spaţiul de adrese al procedurii apelante. Procedura remote nu poate atribui o semnificaţie acestui pointer şi nu va avea acces la acest spaţiu de adrese. Aşadar transmiterea unui pointer pe post de parametru nu este posibilă. Fiecare metodă RPC are o listă cu tipurile de date care pot fi transmise prin reţea.

Fişierul `mon_xdr.c` generat în urma compilării este:

```
/* Please do not edit this file. It was generated using rpcgen. */

#include "mon.h"

bool_t xdr_Data (XDR *xdrs, Data *objp) {

        register int32_t *buf;

        if (!xdr_int (xdrs, &objp->result))
                return FALSE;
        if (!xdr_int (xdrs, &objp->whatToUse))
                return FALSE;
        if (!xdr_string (xdrs, &objp->hostName, 1024))
                return FALSE;
        if (!xdr_string (xdrs, &objp->osType, 128))
                return FALSE;
        if (!xdr_string (xdrs, &objp->osVersion, 32))
                return FALSE;
        if (!xdr_array (xdrs, (char **)&objp->cpuFreq.cpuFreq_val, (u_int *)
&objp->cpuFreq.cpuFreq_len, 65536, sizeof (float), (xdrproc_t) xdr_float))
                return FALSE;
        if (!xdr_float (xdrs, &objp->load5))
                return FALSE;
        if (!xdr_float (xdrs, &objp->memSwap))
                return FALSE;
        if (!xdr_float (xdrs, &objp->memFree))
                return FALSE;
        if (!xdr_double (xdrs, &objp->diskFree))
                return FALSE;
```

```
        return TRUE;
}

bool_t xdr_ConfigData (XDR *xdrs, ConfigData *objp) {

        register int32_t *buf;

        if (!xdr_int (xdrs, &objp->result))
                return FALSE;
        if (!xdr_int (xdrs, &objp->monitoringInterval))
                return FALSE;
        if (!xdr_int (xdrs, &objp->whatToUse))
                return FALSE;
        return TRUE;
}
```

Implementarea funcţiilor pentru servere (atât server-ul studentului cât şi server-ul facultăţii) este următoarea (funcţiile pentru obţinerea parametrilor, aşa cum au fost prezentaţi nu sunt redate aici):

```
ConfigData* read_config_data_1_svc(void* dummy1, struct svc_req* dummy2) {
        static ConfigData result;

        LOG("Reading config data\n");
        ReadConfigData(&result);
        LOG("Config data read\n");

        return &result;
}


Data* read_data_1_svc(int* whatToUse, struct svc_req* dummy) {
        static Data result;

        // detect operating system type (only done once)
        if(OSType == OS_UNKNOWN)
        {
                LOG("Detecting OS type\n");
                DetectOSType();
                LOG("OS type detected\n");
        }

        // free old result
        xdr_free((xdrproc_t)xdr_Data, (char*)&result);

        memset(&result, 0, sizeof(result));
        result.whatToUse = *whatToUse;

        LOG("Reading data %X\n", *whatToUse);
        result.result = ReadData(&result);
        LOG("Data read\n");

        // strings must not be NULL
        if(!result.hostName)
                result.hostName = strdup("N/A");
        if(!result.osType)
                result.osType = strdup("N/A");
        if(!result.osVersion)
                result.osVersion = strdup("N/A");
```

```
        return &result;
}

int* write_data_1_svc(Data* data, struct svc_req* dummy) {
        static int result = 0;

        LOG("Writing data %X\n", data->whatToUse);
        result = WriteData(data);
        LOG("Data written\n");

        return &result;
}
```

Implementarea programului client este următoarea:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <unistd.h>
#include "mon.h"
#include "common.h"

#define LOCAL_SERVER "localhost"

int main(int argc, char* argv[]) {
        CLIENT*         clLocal;
        CLIENT*         clRemote;
        char*           server;
        Data*           data;
        ConfigData*     configData;
        int*            result;

        if(argc != 2) {
                printf("Usage: %s server\n", argv[0]);
                exit(1);
        }

        server = argv[1];

        clLocal = clnt_create(LOCAL_SERVER, mon, monVERS, "tcp");

        if(clLocal == NULL) {
                clnt_pcreateerror(LOCAL_SERVER);
                exit(1);
        }

        configData = read_config_data_1(NULL, clLocal);

        if(configData == NULL) {
                clnt_perror(clLocal, LOCAL_SERVER);
                exit(1);
        }

        if(configData->result == RES_ERROR) {
                return 1;
        }

        clRemote = clnt_create(server, mon, monVERS, "tcp");

        if(clRemote == NULL) {
                printf("Could not connect to remote server\n");
```

```
                clnt_pcreateerror(server);
                exit(1);
        }

        for(;;) {
                data = read_data_1(&configData->whatToUse, clRemote);

                if(data == NULL) {
                        clnt_perror(clRemote, server);
                        exit(1);
                }

                if(data->result == RES_ERROR) {
                        printf("Remote server could not read data\n");
                        return 1;
                }

                result = write_data_1(data, clLocal);

                if(data == NULL) {
                        clnt_perror(clLocal, LOCAL_SERVER);
                        exit(1);
                }

                if(data->result == RES_ERROR) {
                        return 1;
                }

                sleep(configData->monitoringInterval);
        }
}
```

2.8. Aplicaţie practică

Vă propunem scrierea unei aplicaţii unei aplicaţii client-server bazată pe RPC în care un client face o cerere către un server pentru primirea unui număr prim. Pentru realizarea acestei aplicaţii parcurgeţi paşii:

Task1.

Scrieţi descrierea protocolului prin intermediul unui fişier `prime.x`. Protocolul trebuie descris astfel încât clientul să trimită către server o structură care conţine numele clientului (un şir de caractere de maxim 32 de caractere) şi un număr natural `n`. Serverul va genera cel mai apropiat număr natural prim apropiat de `n`, va răspunde clientului şi va scrie într-un fişier un mesaj de log.

Task2.

Scrieţi aplicaţia server care conţine o funcţie care testează că un număr este prim, o funcţie care determină cel mai apropiat număr prim de `n` (parametrul primit de la client). Funcţia întoarce ca rezultat acest număr prim.

Task3.

Scrieţi aplicaţia client care preia din linia de comandă, pe lângă adresa server-ului, numele său şi numărul n. Clientul va face o singură cerere către server şi va afişa pe ecran rezultatul primit.

Task4.

Scrieţi un fişier `makefile` pentru compilarea întregii aplicaţiei. Rulaţi aplicaţia şi urmăriţi fişierul de log generat de server în urma cererilor venite de la mai mulţi clienţi. Verificaţi că informaţiile logate de server sunt identice cu cele afişate de clienţi ca rezultate.

2.9. Probleme propuse

Problema 1.

De pe unul dintre sistemele conectate la o reţea se doreşte aflarea utilizatorilor activi de pe oricare sistem interconectat la reţea respectivă. Acest lucru se poate face prin invocarea unei comenzi de sistem. Să se rezolve problema folosind rpcgen. Să se scrie fişierul `makefile` care construieşte programele client şi server.

Problema 2.

Folosind rpcgen să se scrie programele client şi server pentru gestionarea unui arbore cu acces multiplu. Un client va defini funcţii de inserare şi eliminare nod în/din arbore, precum şi o funcţie de afişare arbore. Mai mulţi clienţi pot accesa arborele memorat de server. Serverul având o memorie limitată, va limita dimensiunea maximă a arborelui. Rezolvarea se va face folosind programul rpcgen.

Problema 3.

Să se implementeze un shell remote folosind o aplicaţie client-server bazată de RPC. Clientul trimite o comandă unui calculator din reţea prin intermediul unui şir de caractere şi un număr care reprezintă numărul maxim de caractere pe care clientul îl doreşte din răspunsul server-ului. Server-ul execută comanda şi răspunde clientului.