

# Paradigma RPC

**Lenuta Alboaie**  
**adria@info.uaic.ro**

# Cuprins

- **Remote Procedure Call (RPC)**
  - Preliminarii
  - Caracterizare
  - XDR
  - Functionare
  - Implementari
  - Utilizari

# Preliminarii

- **Proiectarea aplicatiilor distribuite**
  - Orientata pe protocol – *socket*-uri
    - Se dezvoltă protocolul, apoi aplicațiile care îl implementează efectiv
  - Orientata pe funcționalitate – **RPC**
    - Se creează aplicațiile, după care se divid în componente și se adaugă protocolul de comunicație între componente

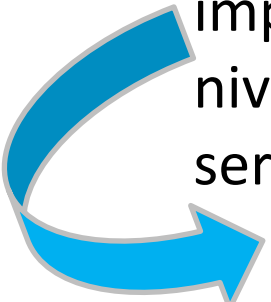
# RPC | Caracterizare

- **Idee:** In loc de accesarea serviciilor la distanta prin trimiterea si primire de mesaje, clientul **apeleaza o procedura care va fi executata pe alta masina**
- **Efect:** RPC “ascunde” existenta retelei de program
  - Mecanismul de *message-passing* folosit in comunicarea in retea este ascuns fata de programator
  - Programatorul nu trebuie sa mai deschida o conexiune, sa citeasca sau sa scrie date, sa inchida conexiunea etc.
- Este un instrument de programare mai simplu decat interfata *socket* BSD

# RPC | Caracterizare

- O aplicatie RPC va consta dintr-un **client** si un **server**, serverul fiind localizat pe masina pe care se executa procedura
- La realizarea unui apel la distanta, parametrii procedurii sunt transferati prin retea catre aplicatia care executa procedura; dupa terminarea executiei procedurii rezultatele sunt transferate prin retea aplicatiei client
- Clientul si serverul → procese pe masini diferite

# RPC | Caracterizare

- **RPC realizeaza comunicarea dintre client si server prin socket-uri TCP/IP (uzual, UDP), via doua interfete *stub* (*ciot*)**
    - Obs. Pachetul RPC (*client stub* si *server stub* | *skeleton*) ascunde toate detaliile legate de programarea in retea
  - **RPC implica urmatoorii pasi:**
    1. Clientul invoca procedura *remote*
      - Se apeleaza o procedura locala, numita *client stub* care impacheteaza argumentele intr-un mesaj si il trimite nivelului transport, de unde este transferat la masina server *remote*
-  *Marshalling (serializare)* = mecanism ce include codificarea argumentelor intr-un format standard si impachetarea lor intr-un mesaj

# RPC | Caracterizare

- **RPC** implica urmatorii **pasi**:

## 2. Server-ul:

- nivelul transport trimite mesajul catre *server stub*, care despacheteaza parametri si apeleaza functia dorita;
- Dupa ce functia returneaza, *server stub* preia valorile intoarse le impacheteaza (*marshalling*) intr-un mesaj si le trimite la *client stub*

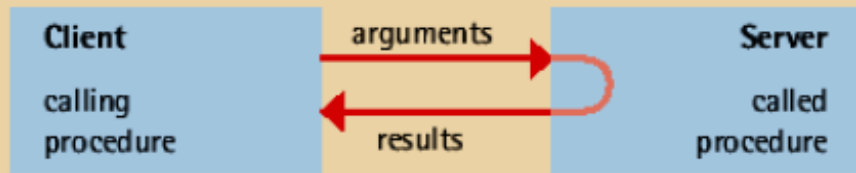


*un-marshalling (deserializare)* = decodificare

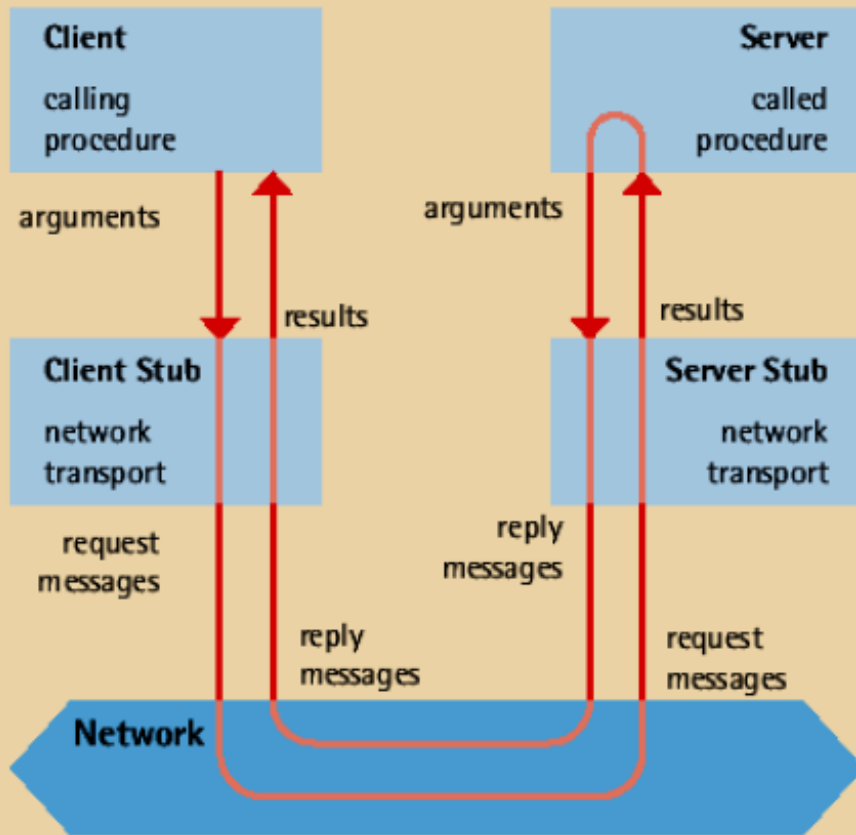
- ## 3. *Client stub* preia valorile primite si le returneaza aplicatiei client

# RPC | Caracterizare

- Interfetele ciot implementeaza protocolul RPC
- Diferente fata de apeluri locale:
  - Performanta poate fi afectata de timpul de transmisie
  - Tratarea erorilor este mai complexa
  - Locatia server-ului trebuie sa fie cunoscuta (Identificarea si accesarea procedurii la distanta)
  - Poate fi necesara autentificarea utilizatorilor



Local Procedure Call



Remote Procedure Call



# RPC | Caracterizare

- Procedurile ciot se pot genera automat, dupa care se “leaga” de programele client si server
- Ciotul serverului asculta la un port si realizeaza invocarea rutinelor printr-o interfata de apel de proceduri locale
- Clientul si serverul vor comunica prin mesaje, printr-o reprezentare independenta de retea si de sistemul de operare:

**External Data Representation (XDR)**

# RPC | Caracterizare

- **External Data Representation (XDR)**

XDR definește numeroase tipuri de date și modul lor de transmisie în mesajele RPC (RFC 1014)

– Tipuri uzuale:

- Preluate din C: int, unsigned int, float, double, void,...
- Suplimentare: string, fixed-length array, variable-length array, ...

– Funcții de conversie (**rpc/xdr.h**)

- **xdrmem\_create()** – asociază unei zone de memorie un flux de date RPC
- **xdr\_numetip()** – realizează conversia datelor

# RPC|Caracterizare

- External Data Representation (XDR)

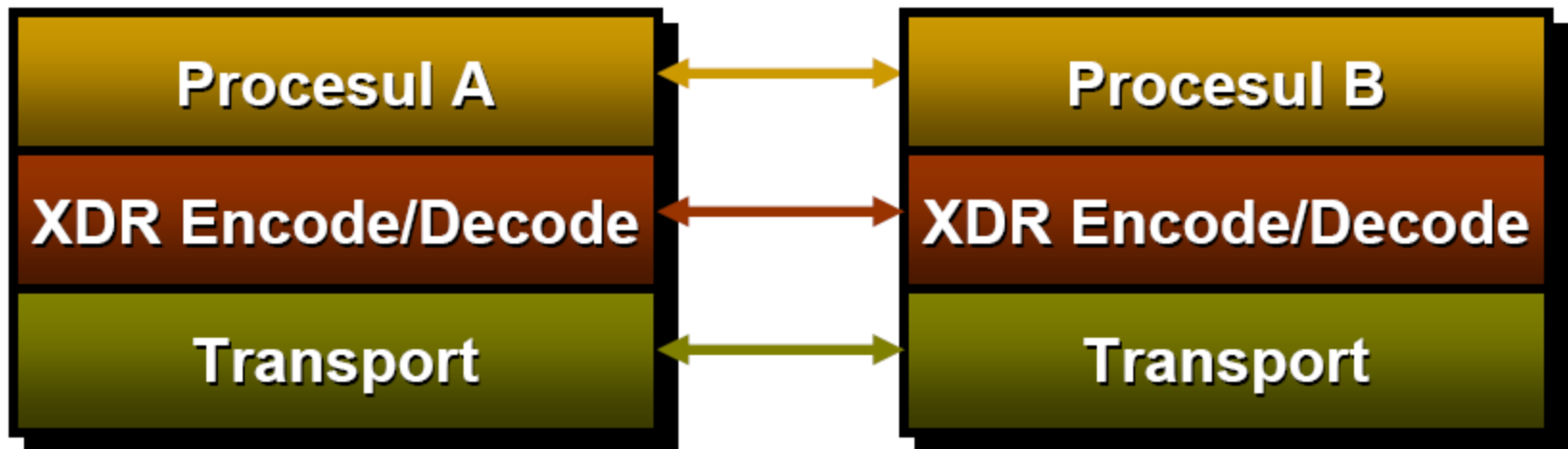
## Exemplu

```
#include <rpc/xdr.h>
#define BUFSIZE 400 /* lungimea zonei de memorie */
/* conversia unui intreg in format XDR */
...
XDR *xdr; /* zona de memorie XDR */
char buf[BUFSIZE];
int intreg;
...
xdrmem_create (xdr, buf, BUFSIZE, XDR_ENCODE);
...
intreg = 33;
xdr_int (xdr, &intreg);
...
```

Inlocuit la celalalt capat al  
comunicatiei cu **XDR\_DECODE**

# RPC | Caracterizare

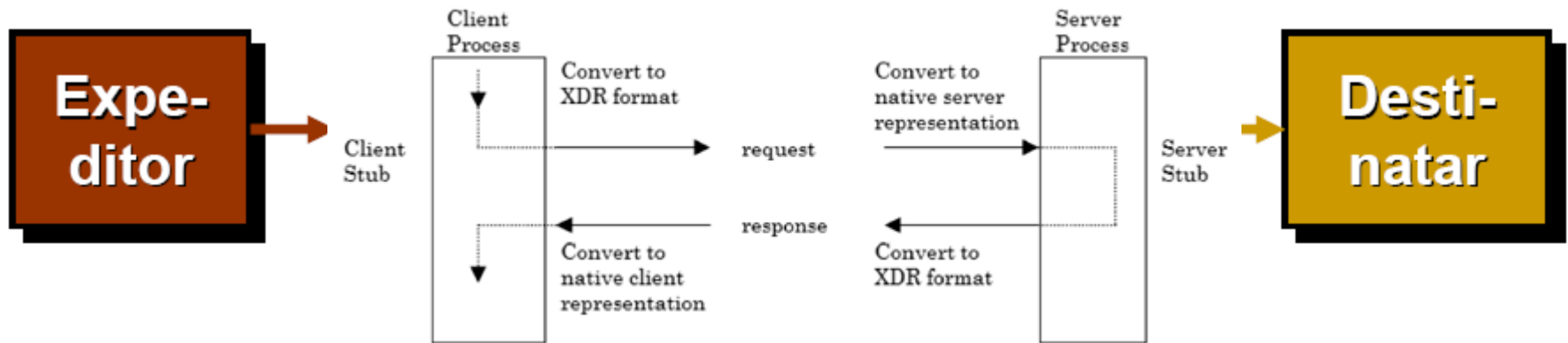
- **External Data Representation (XDR)**
  - Poate fi vazut ca nivel suplimentar intre nivelul transport si nivelul aplicatie
  - Asigura conversia simetrica a datelor client si server



# RPC | Caracterizare

## External Data Representation (XDR)

- Activitatea de codificare/decodificare



- In prezent, poate fi înlocuit de reprezentari XML-RPC sau SOAP sau JSON-RPC (in contextul serviciilor Web)



vezi cursul de Tehnologii Web!

# RPC | Functionare

Context:

- Un serviciu de retea este identificat de portul la care exista un *daemon* asteptind cereri
- Programele server RPC folosesc porturi efemere



De unde stie clientul unde sa trimita cererea?

**Portmapper** = serviciu de retea responsabil cu asocierea de servicii la diferite porturi

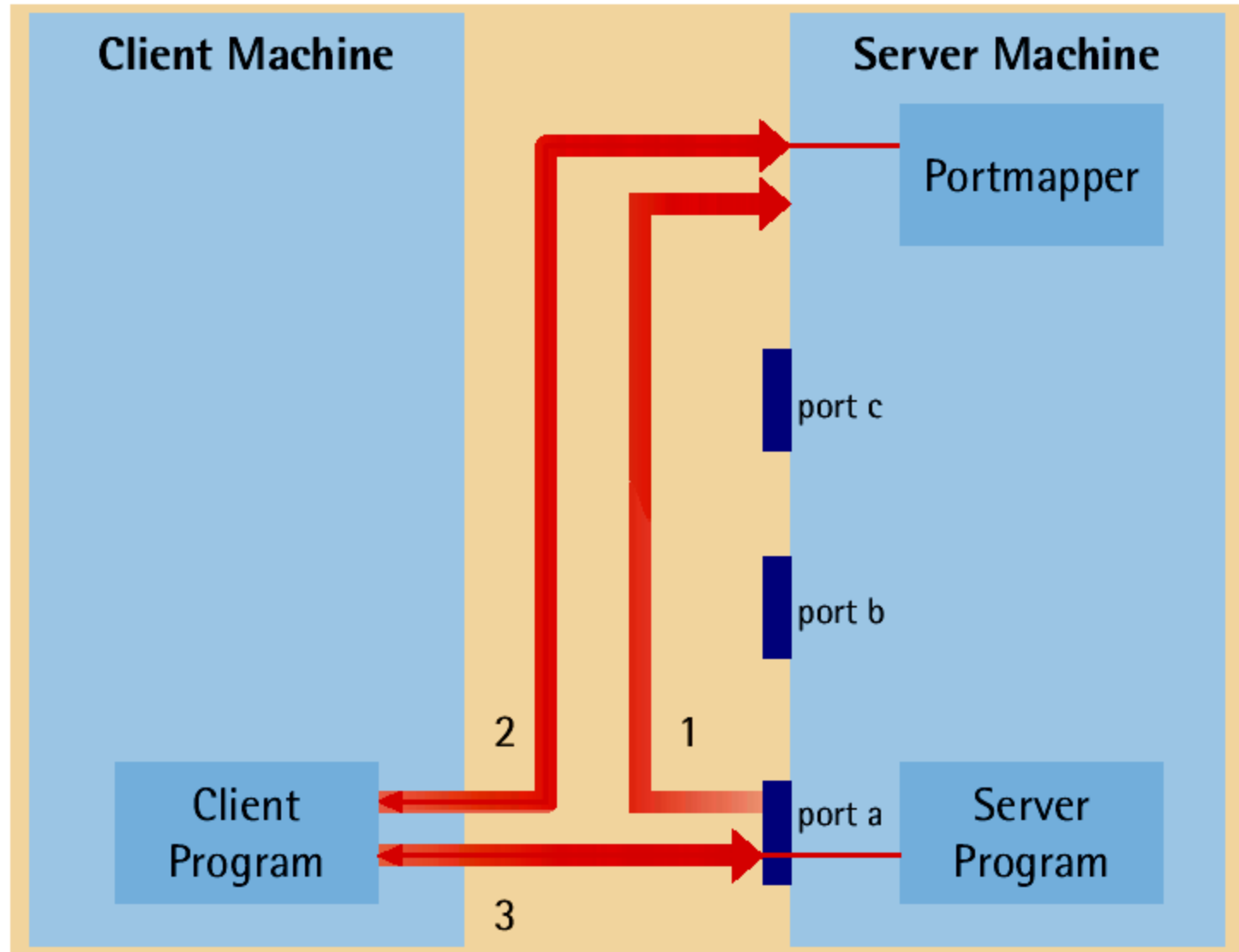
=> Numerele de port pentru un anumit serviciu nu sunt fixe

- Este disponibil la portul 111 (*well-known port*)

```
[adria@thor ~] $ rpcinfo -p
  program vers proto  port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp   56660  status
    100024    1   tcp   48918  status
```

# RPC | Functionare

## Mecanism general



[Rețele de calculatoare –  
curs 2007-2008, Sabin Buraga]

# RPC | Functionare

## Mecanism general:

**Pas 1:** Se determina adresa la care serverul va oferi serviciul

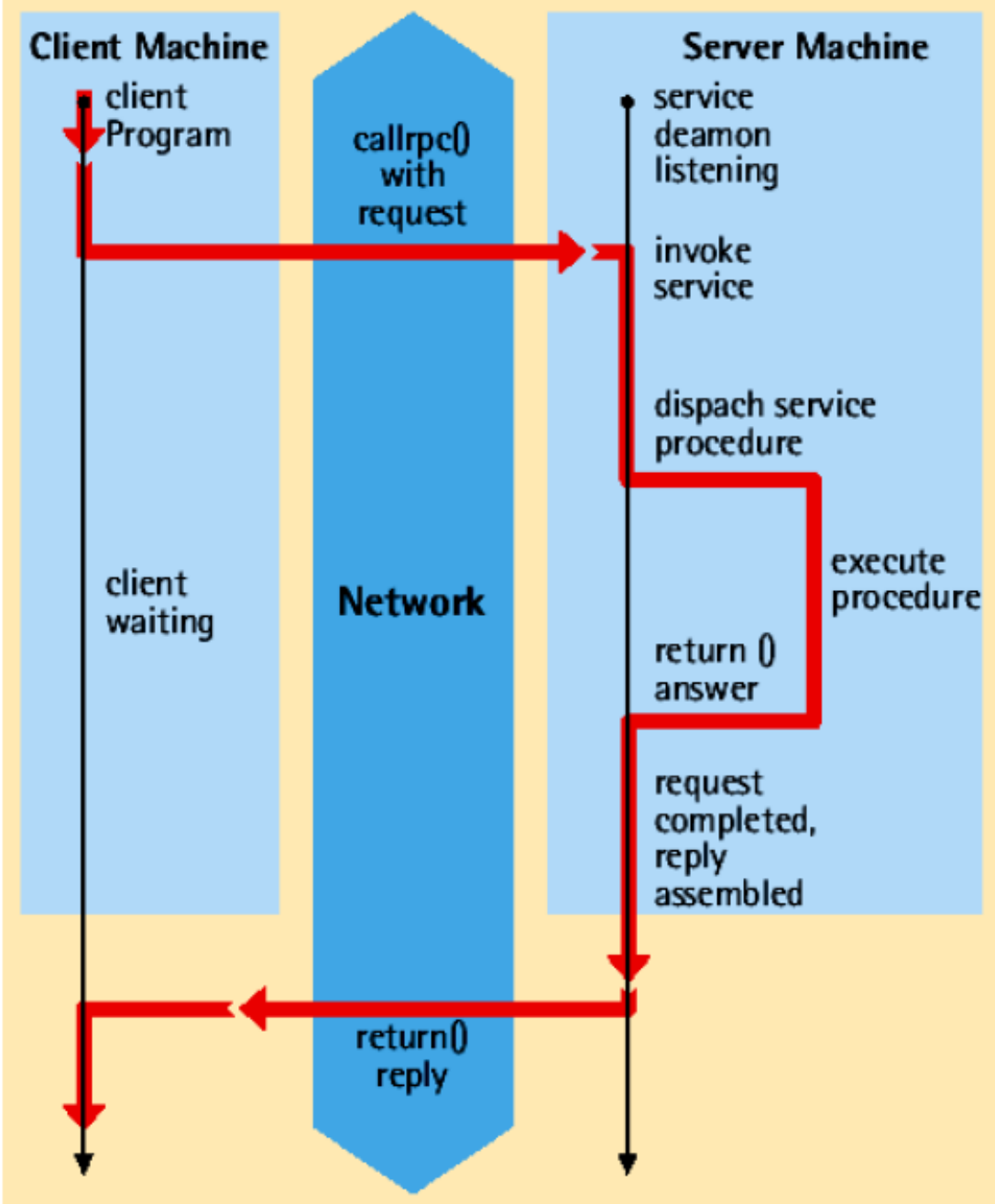
- La initializare, serverul stabileste si inregistreaza via *portmapper* portul la care va oferi serviciul (portul **a**)

**Pas 2:** Clientul consulta *portmapper*-ul de pe masina serverului pentru a identifica portul la care trebuie sa trimita cererea RPC

**Pas 3:** Clientul si serverul pot comunica pentru a realiza executia procedurii la distanta

- Cererile si raspunsurile sunt (de)codificate prin XDR



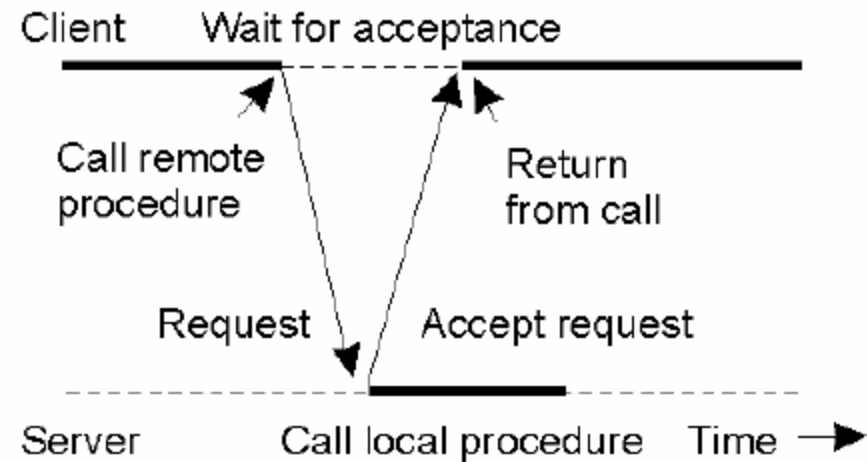
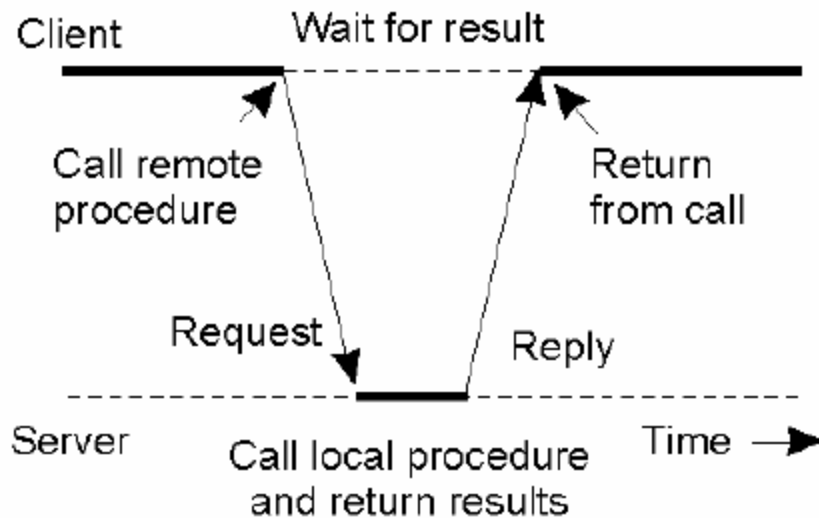


# RPC | Functionare

- Atunci cind un server furnizeaza mai multe servicii, este de obicei folosita o rutina *dispatcher*
- *Dispatcher-ul* identifica cererile specifice si apeleaza procedura corespunzatoare, dupa care rezultatul este trimis inapoi clientului pentru a-si continua executia

# RPC | Functionare

- Transferurile de date RPC pot fi:
  - Sincrone
  - Asincrone



# RPC | Implementare

- **Open Network Computing RPC (ONC RPC)** - cea mai raspindita implementare in mediile Unix (Sun Microsystems)
  - RFC 1057
  - Interfata RPC este structurata pe 3 niveluri:
    - **Superior:** independent de sistem, hardware sau retea
      - Exemplu: man **rcmd** -> *routines for returning a stream to a remote command ....*
    - **Intermediar:** face apel la functiile definite de biblioteca RPC:
      - **registorpc()** – inregistreaza o procedura spre a putea fi executata la distanta
      - **callrpc()** – apeleaza o procedura la distanta
      - **svc\_run()** – ruleaza un serviciu RPC
    - **Inferior:** da posibilitatea de a controla in detaliu mecanismele RPC (e.g. alegerea modului de transport al datelor etc)

# RPC | Implementare

- Open Network Computing RPC (ONC RPC)
  - Procedurile la distanta se vor include intr-un **program la distanta** - unitate software care se va executa pe o masina la distanta
  - Fiecare program la distanta corespunde unui server: putind contine proceduri la distanta + date globale; procedurile pot partaja date comune;
  - Fiecare program la distanta se identifica printr-un identificator unic stocat pe 32 biti; Conform implementarii Sun RPC avem urmatoarele valori ale identificatorilor:
    - 0x00 00 00 00 – 0x1F FF FF FF - aplicatiile RPC ale sistemului
    - 0x20 00 00 00 – 0x3F FF FF FF – programele utilizator
    - 0x40 00 00 00 – 0x5F FF FF FF – identificatori temporari
    - 0x60 00 00 00 – 0xFF FF FF FF – valori rezervate
  - Fiecare procedura (din cadrul unui program) este identificata printr-un index (1..n)

# RPC | Implementare

- Open Network Computing RPC (ONC RPC)


Exemple:

- 10000 meta-serverul *portmapper*
- 10001 pentru *rstatd* care ofera informatii despre sistemul *remote*; se pot utiliza procedurile *rstat()* sau *perfmeter()*
- 10002 pentru *rusersd* ce furnizeaza informatii despre utilizatorii conectati pe masina la distanta
- 10003 serverul *nfs* – ce ofera acces la sistemul de fisiere in retea *NFS (Network File System)*

# RPC | Implementare

- Open Network Computing RPC (ONC RPC)

Fiecare program la distanta are asociat un numar de versiune

- 
- Initial versiunea 1
  - Urmatoarele versiuni se identifica in mod unic prin alte numere de versiune

Se ofera posibilitatea de a schimba detaliile de implementare sau extinderea capabilitatilor aplicatiei fara a asigura un alt identificator unui program

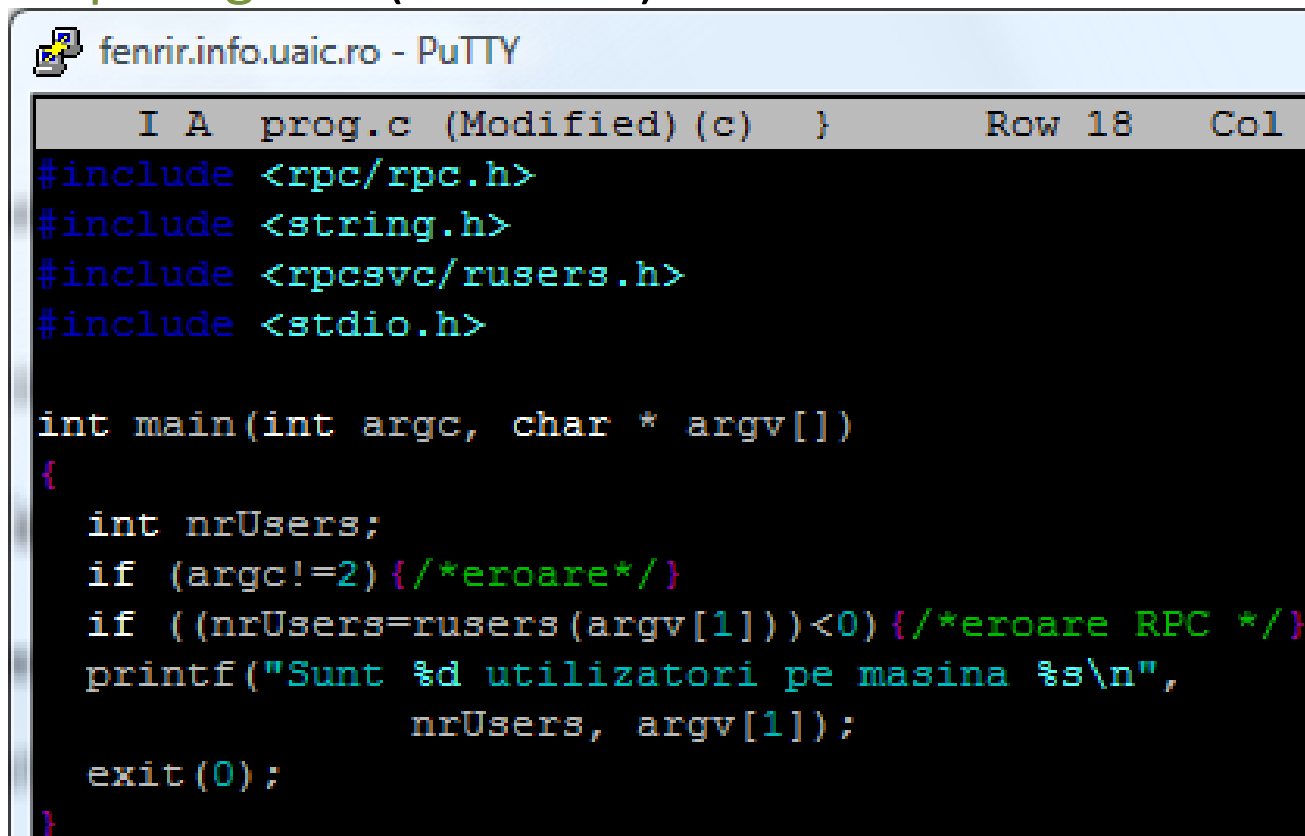
Un program la distanta este un 3-uplu de forma:

`<id_Program, versiune, idex_procedura>`

# RPC|Implementare

- Open Network Computing RPC (ONC RPC)

**Programare  
de nivel  
inalt:**



```
fenrir.info.uaic.ro - PuTTY
I A  prog.c (Modified) (c)  }      Row 18  Col
#include <rpc/rpc.h>
#include <string.h>
#include <rpcsvc/rusers.h>
#include <stdio.h>

int main(int argc, char * argv[])
{
    int nrUsers;
    if (argc!=2){/*eroare*/
    if ((nrUsers=rusers(argv[1]))<0){/*eroare RPC */}
    printf("Sunt %d utilizatori pe masina %s\n",
            nrUsers, argv[1]);
    exit(0);
}
```

Compilare: gcc prog.c -lrpcsvc -o prog

Executie: ./prog fenrir.infoiasi.ro

# RPC | Implementare

- Open Network Computing RPC (ONC RPC)

**Programare la nivel intermediar:**

```
callrpc (char *host, /* numele serverului */  
          u_long prognum, /* numarul programului server */  
          u_long versnum, /* numarul de versiune a serv.*/  
          u_long procnum, /* numarul procedurii */  
          xdrproc_t inproc, /* fol. pentru codificare XDR*/,  
          char *in, /* adresa argumentelor procedurii*/,  
          xdrproc_t outproc, /* fol. pentru decodificare */,  
          char *out, /* adresa de plasare a rezultatelor*/  
          );
```

Apelata  
de  
clientul  
RPC



# RPC | Implementare

- Open Network Computing RPC (ONC RPC)

Programare la nivel intermediar:

**registerrpc(**

Apelata  
de  
serverul  
RPC


```
u_long prognum /* numarul programului server */,  
u_long versnum /* numarul de versiune a serv*/,  
u_long procnum /* numarul procedurii */,  
void *(*procname)*() /* numele functiei remote */,  
xdrproc_t inproc /* fol. pt. decodificarea param. */,  
xdrproc_t outproc /* fol. pt. codificarea result. */  
);
```

# RPC | Implementare

- Open Network Computing RPC (ONC RPC)

**Programare la nivel intermediar:**

`svc_run ()`



Apelata de serverul RPC,  
reprezinta *dispatcher*-ul

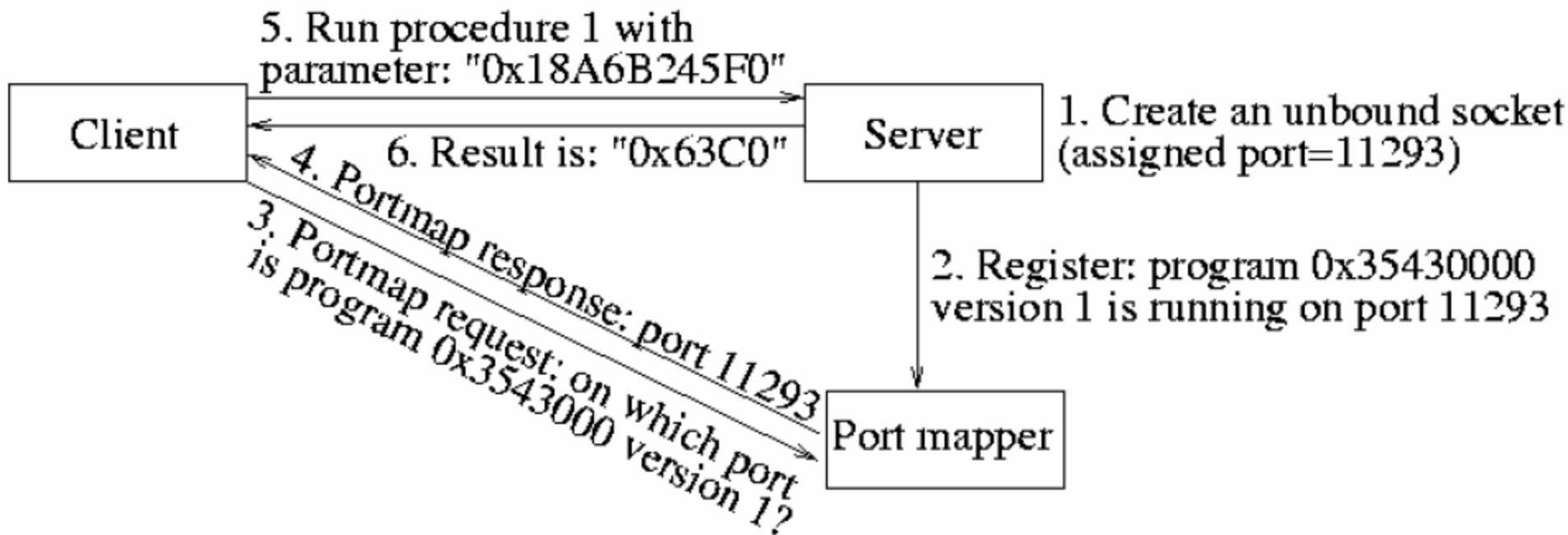
- Se asteapta venirea de cereri RPC, apoi se apeleaza folosindu-se `svc_getreq()` procedura corespunzatoare

Obs. Functiile de nivel intermediar utilizeaza doar UDP

# RPC|Implementare

- Open Network Computing RPC (ONC RPC)

**Programare la nivel inferior:**



# RPC | Implementare

- Open Network Computing RPC (ONC RPC)

Realizarea de aplicatii RPC cu **rpcgen**

- Se creaza un fisier cu specificatii RPC (Q.x)
  - Declaratii de constante utilizate de client si server
  - Declaratii de tipuri de date globale
  - Declaratii de programe la distanta, proceduri, tipuri de parametri, tipul rezultatului, identificatorul unic de program
- Programul server.c care contine procedurile
- Programul client.c care invoca procedurile

Pentru server: **gcc server.c Q\_svc.c Q\_xdr.c -o server**

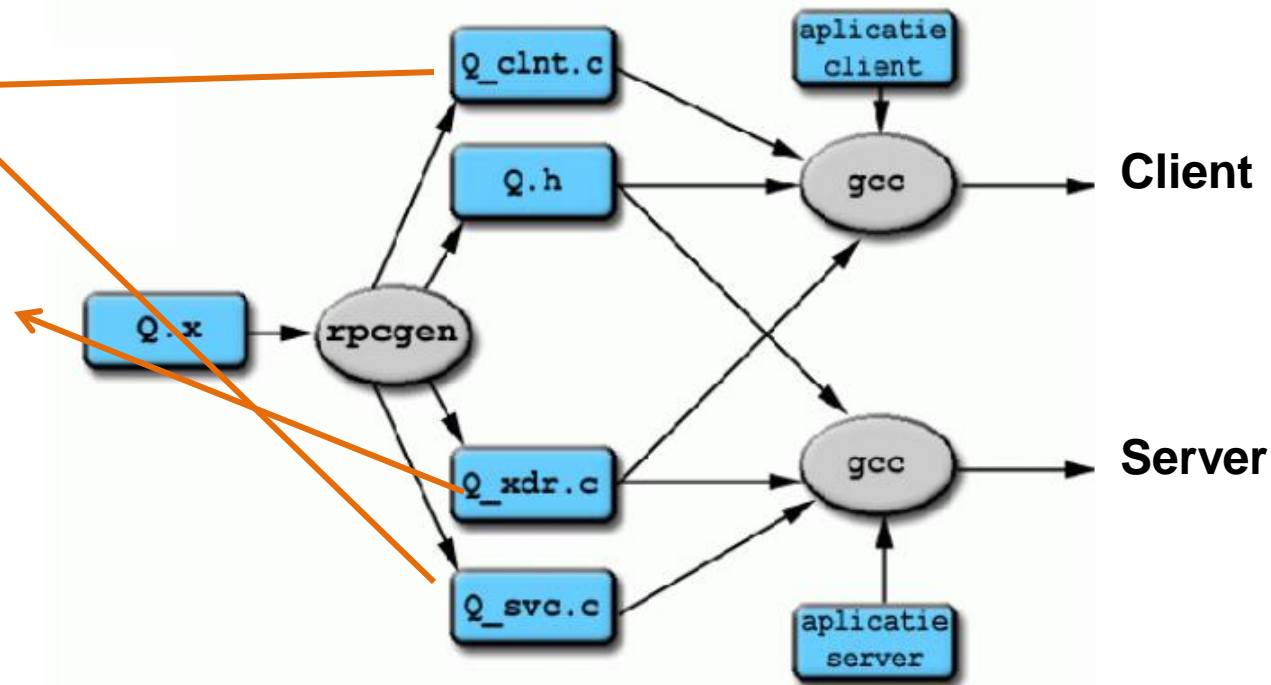
Pentru client: **gcc client.c Q\_clnt.c Q\_xdr.c -o client**

# RPC | Implementare

- Open Network Computing RPC (ONC RPC)

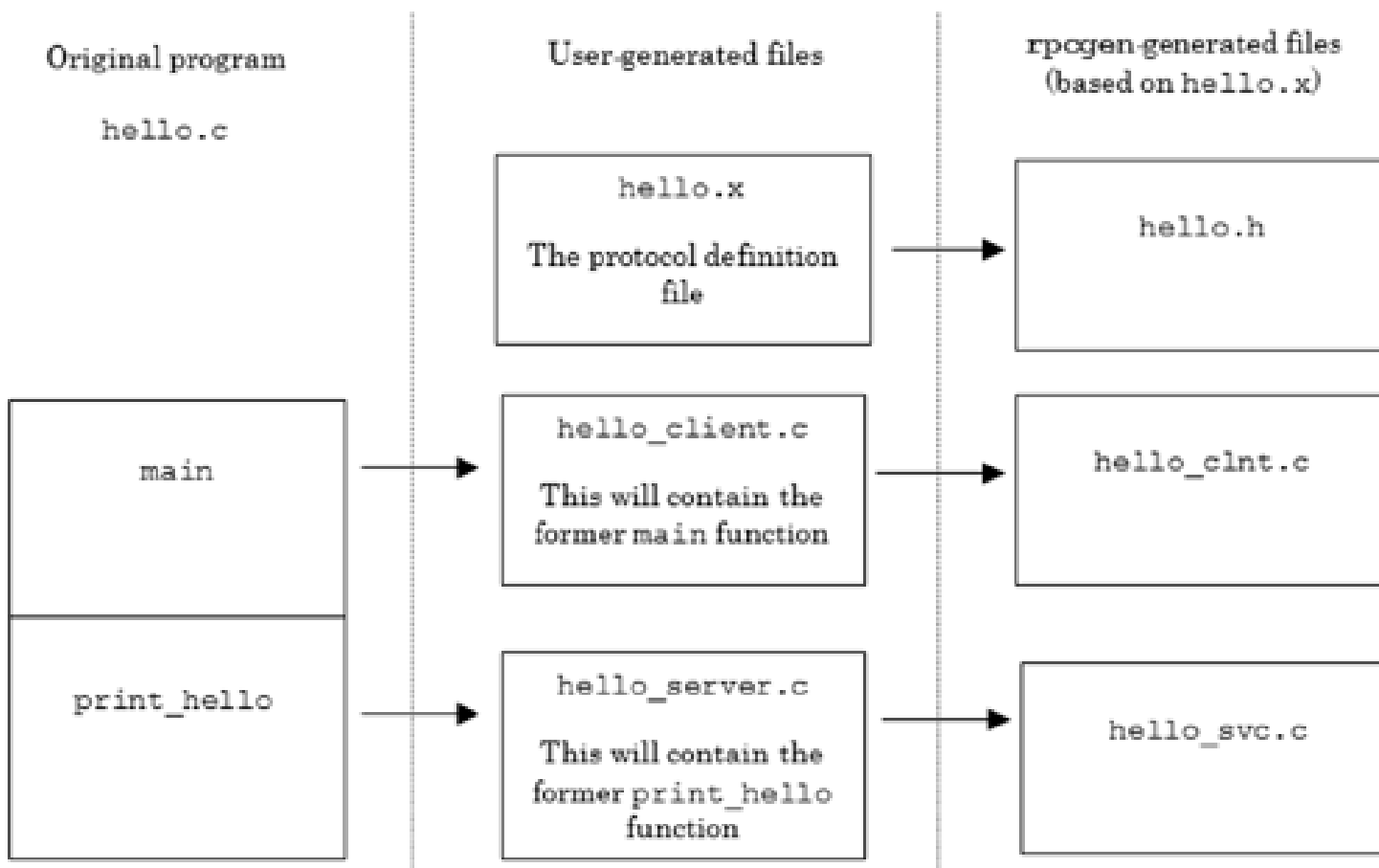
În implementarea unei aplicații RPC se utilizează utilitarul **rpcgen**

- Generează *client stub* și *server stub*
- Generează funcțiile de codificare și decodificare XDR
- Generează rutina *dispatcher*



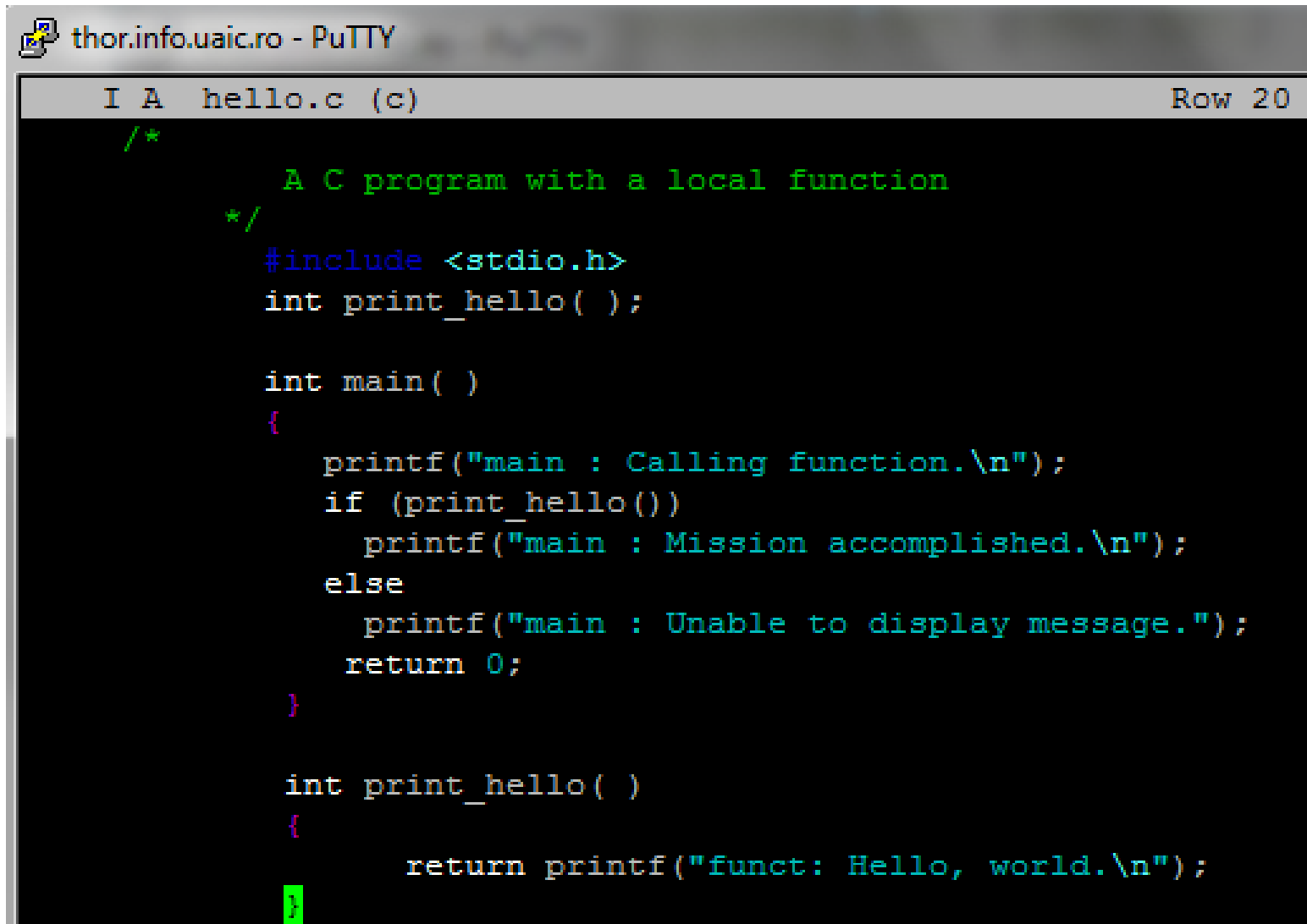
# RPC | Implementare

## Client-server files and relationships.



[Interprocess Communications in Linux, J.S. Gray]

# RPC|Implementare



```
thor.info.uaic.ro - PuTTY
I A hello.c (c) Row 20
/*
    A C program with a local function
*/
#include <stdio.h>
int print_hello( );

int main( )
{
    printf("main : Calling function.\n");
    if (print_hello())
        printf("main : Mission accomplished.\n");
    else
        printf("main : Unable to display message.");
    return 0;
}

int print_hello( )
{
    return printf("funct: Hello, world.\n");
}
```

[Interprocess Communications in Linux, J.S. Gray]

# RPC|Implementare

```
thor.info.uaic.ro - PuTTY  
[adria@thor ~/rpc] $ ls  
hello.c  hello.x  
[adria@thor ~/rpc] $ cat hello.x  
program DISPLAY_PRG {  
    version DISPLAY_VER {  
        int print_hello( void ) = 1;  
    } = 1;  
} = 0x200000001;  
  
[adria@thor ~/rpc] $ rpcgen -C hello.x  
[adria@thor ~/rpc] $ ls -al  
total 28  
drwxr-xr-x  2 adria profs 4096 2011-12-12 17:16 .  
drwx--x--x 49 adria profs 4096 2011-12-12 17:15 ..  
-rw-r--r--  1 adria profs  777 2011-12-12 17:14 hello.c  
-rw-r--r--  1 adria profs  545 2011-12-12 17:16 hello_clnt.c  
-rw-r--r--  1 adria profs  711 2011-12-12 17:16 hello.h  
-rw-r--r--  1 adria profs 2163 2011-12-12 17:16 hello_svc.c  
-rw-r--r--  1 adria profs  133 2011-12-12 17:14 hello.x  
[adria@thor ~/rpc] $
```

[Interprocess Communications in Linux, J.S. Gray]



# RPC|Implementare

```
I A hello_client.c (c)      Row 1   Col 1   10:01   Ctrl-K H for help
/*
    The CLIENT program:  hello_client.c
    This will be the client code executed by the local client process.
*/
#include <stdio.h>
#include "hello.h"           /* Generated by rpcgen from hello.x */
int
main(int argc, char *argv[]) {
    CLIENT      *client;
    int         *return_value, filler;
    char        *server;
/*
    We must specify a host on which to run.  We will get the host name
    from the command line as argument 1.
*/
    if (argc != 2) {
        fprintf(stderr, "Usage: %s host_name\n", *argv);
        exit(1);
    }
    server = argv[1];
/*
    Generate the client handle to call the server
*/
    if ((client=clnt_create(server, DISPLAY_PRG, DISPLAY_VER, "tcp")) ==
        clnt_pcreateerror(server);
        exit(2);
    }
    printf("client : Calling function.\n");
    return_value = print_hello_1((void *) &filler, client);
    if (*return_value)
        printf("client : Mission accomplished.\n");
    else
        printf("client : Unable to display message.\n");
    return 0;
}
```

[Interprocess  
Communicati  
ons in Linux,  
J.S. Gray]

# RPC|Implementare

```
I A  hello_server.c (c)          Row 1   Col 1   10:02   Ctrl-K H for help

/*
   The SERVER program: hello_server.c
   This will be the server code executed by the "remote" process
*/
#include <stdio.h>
#include "hello.h"                /* is generated by rpcgen from hello.x */
int *
print_hello_1_svc(void * filler, struct svc_req * req) {
    static int  ok;
    ok = printf("server : Hello, world.\n");
    return (&ok);
}
```

[Interprocess  
Communicati  
ons in Linux,  
J.S. Gray]

# RPC | Implementare

```
[adria@thor ~/html/teach/courses/net/files/NetEx/S11/RPC/tempRPC] $ rpcinfo -p
```

program	vers	proto	port	
100000	2	tcp	111	portmapper
100000	2	udp	111	portmapper
100024	1	udp	56604	status
100024	1	tcp	34914	status

```
[adria@thor ~/html/teach/courses/net/files/NetEx/S11/RPC/tempRPC] $
```

```
~/tempRPC] $ gcc hello_client.c hello_clnt.c -o client
~/tempRPC] $ gcc hello_server.c hello_svc.c -o server
~/tempRPC] $ ./server
```

```
[adria@thor ~/html/teach/courses/net/files/NetEx/S11/RPC/tempRPC] $ rpcinfo -p
```

program	vers	proto	port	
100000	2	tcp	111	portmapper
100000	2	udp	111	portmapper
100024	1	udp	56604	status
100024	1	tcp	34914	status
536870913	1	udp	37547	
536870913	1	tcp	43833	

```
[adria@thor ~/html/teach/courses/net/files/NetEx/S11/RPC/tempRPC] $
```

```
[adria@thor ~/html/teach/courses/net/files/NetEx/S11/RPC/tempRPC] $ ./client 127.0.0.1
client : Calling function.
client : Mission accomplished.
```

```
[adria@thor ~/html/teach/courses/net/files/NetEx/S11/RPC/tempRPC] $ ./server
server : Hello, world.
```

# RPC|Implementare

- Alte implementari:
  - DCE/RPC (Distributed Computing Environment/RPC)
    - Alternativa la Sun ONC RPC
    - Utilizat si de serverele Windows
  - ORPC (Object RPC )
    - Mesajele de cerere/raspuns la distanta se incapsuleaza in obiecte
    - Descendenti directi:
      - (D)COM (Distributed Component Object Model) & CORBA (Common Object Request Broker Architecture)
      - In Java: RMI (Remote Method Invocation)
      - .Net Remoting , WCF
  - SOAP (Simple Object Access Protocol)
    - XML ca XDR, HTTP ca protocol de transfer
    - Baza de implementare a unei categorii de servicii Web

# RPC | Utilizari

- **Accesul la fisiere la distanta NFS ( Network File System)**
  - Protocol proiectat a fi independent de masina, sistem de operare si de protocol – implementat peste RPC ( ... conventia XDR)
  - Protocol ce permite partajare de fisiere in retea => NFS ofera acces transparent clientilor la fisiere
    - Obs: Diferit fata de FTP (vezi curs anterior)
  - Ierarhia de directoare NFS foloseste terminologia UNIX (arbore, director, cale, fisier etc.)
  - NFS este un protocol => client - **nfs** , server –**nfsd** comunicind prin RPC
  - Modelul NFS
    - Operatii asupra unui fisier la distanta: operatii I/O, creare/redenumire/stergere, stat, listarea intrarilor
    - Comanda **mount** - specifica gazda *remote*, sistemul de fisiere ce trebuie accesat si unde trebuie sa fie localizat in ierarhia locala de fisiere
  - RFC 1094

# RPC | Utilizari

- Accesul la fisiere la distanta **NFS (Network File System)**
  - Este transparent pentru utilizator
  - Clientul NFS trimite o cerere RPC, serverului RPC folosind TCP/IP
    - Obs. NFS a fost folosit predominant cu UDP
  - Serverul NFS primeste cererile la portul 2049, si le trimite la modulul de accesare a fisierelor locale

Obs. Pentru deservirea rapida a clientilor, serverele NFS sunt in general *multi-threading* sau pentru sisteme UNIX care nu sunt *multi-threading*, se creaza si ramin in kernel instante multiple a procesului (*nfsd-uri*)

# RPC | Utilizari

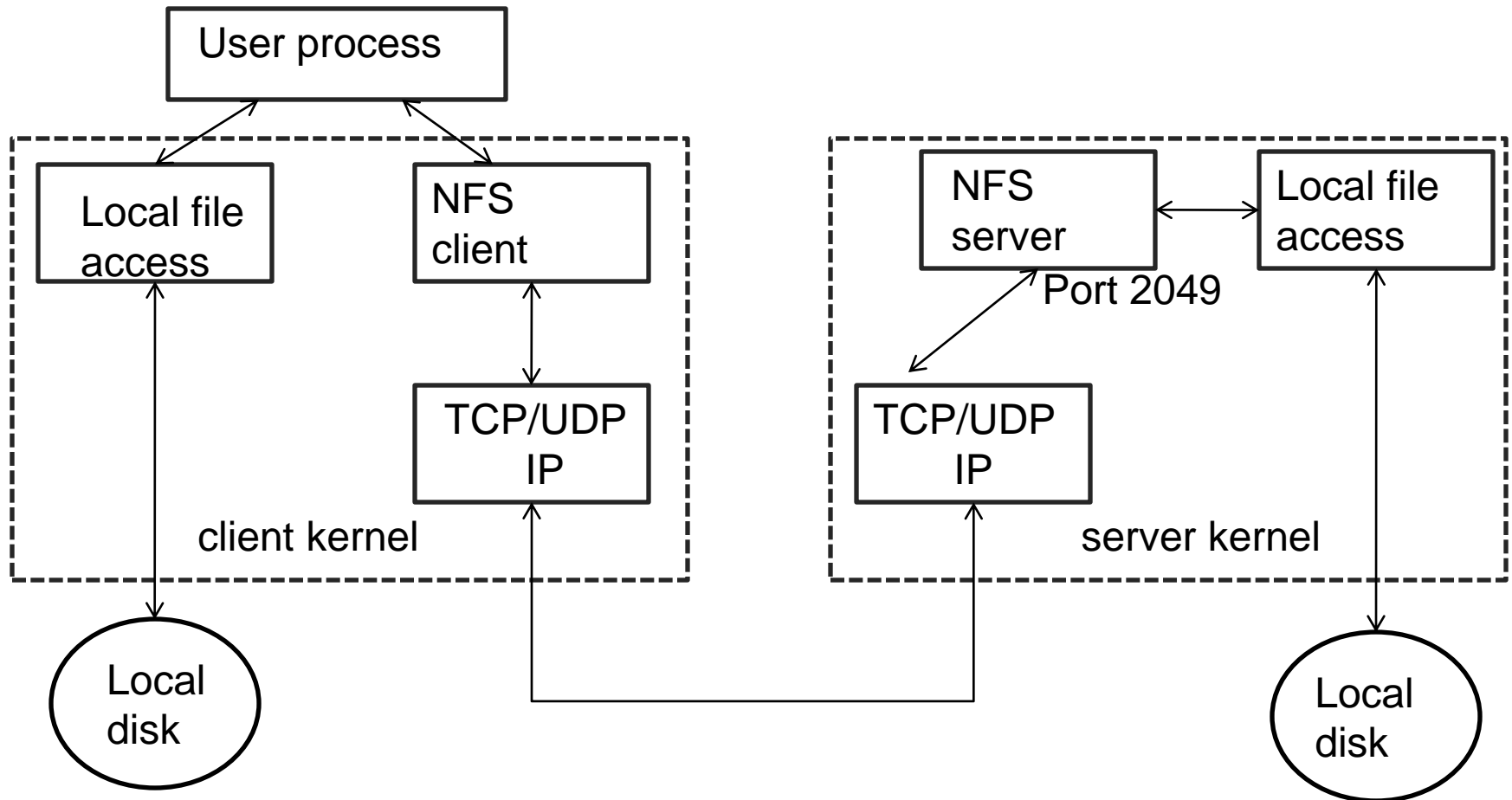


Figura: Arhitectura NFS

# RPC | Utilizari

- Accesul la fisiere la distanta **NFS (Network File System)**
  - (0) este pornit *portmapper*–ul la *boot*-area sistemului
  - (1) daemonul *mountd* este pornit pe server; creaza *end-point*-uri TCP si UDP, le asigneaza porturi efemere si apeleaza la *portmapper* pentru inregistrarea lor
  - (2) se executa comanda *mount* si se face o cerere la *portmapper* pentru a obtine portul serverului *demon* de *mount*
  - (3) *portmapper*–ul intoarce raspunsul
  - (4) se creaza o cerere RPC pentru montarea unui sistem de fisiere
  - (5) serverul returneaza un *file handle* pentru sistemul de fisiere cerut
  - (6) Se asociaza acestui *file handle* un punct de montare local pe client (*filehandle* este stocat in codul clientului NFS, si orice cerere pentru sistemul de fisiere respectiv va utiliza acest *file handle* )



# RPC|Utilizari

- Accesul la fisiere la distanta **NFS (Network File System)**
  - Procesul de montare (protocolul **mount**)
    - Pentru ca un client sa poata accesa fisiere dintr-un sistem de fisiere, clientul trebuie sa foloseasca protocolul **mount**

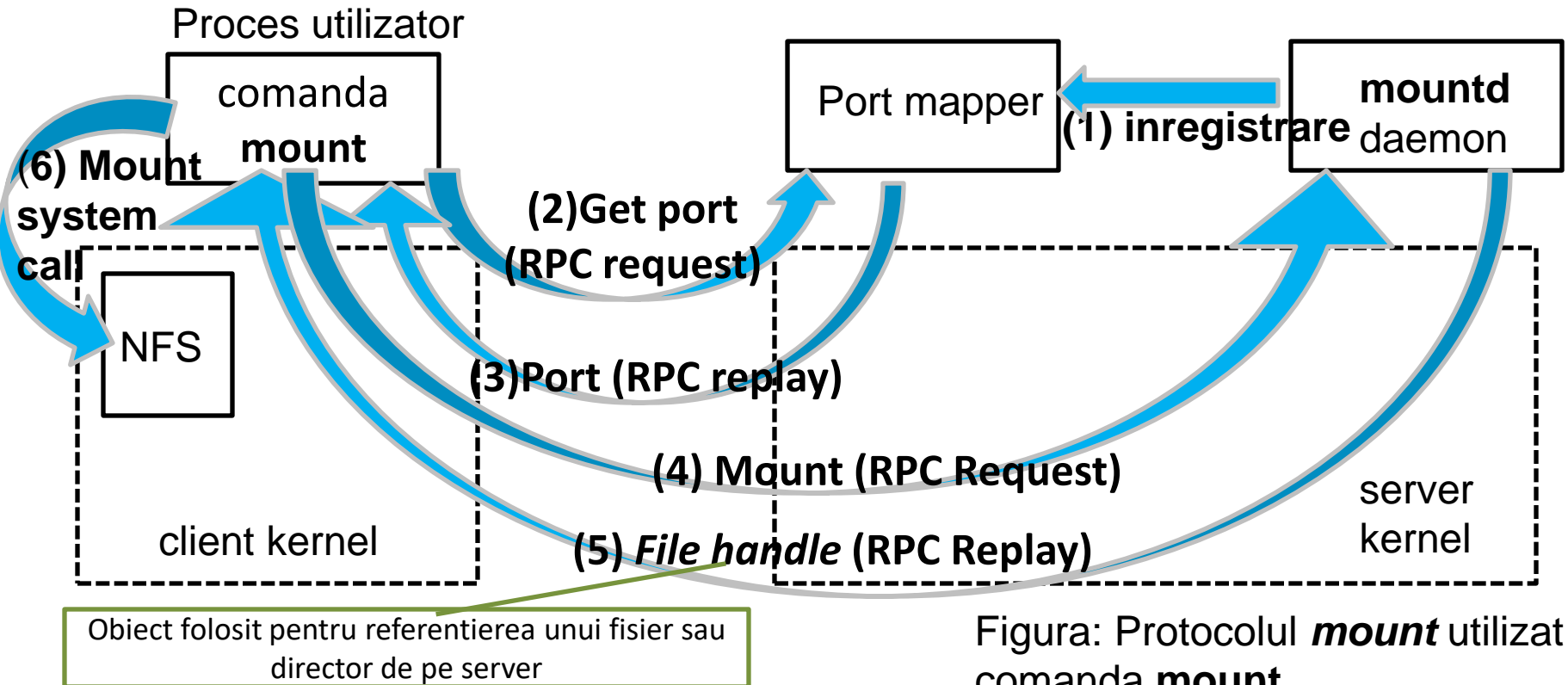


Figura: Protocolul **mount** utilizat de comanda **mount**

# Rezumat

- **Remote Procedure Call (RPC)**
  - Preliminarii
  - Caracterizare
  - XDR
  - Functionare
  - Implementari
  - Utilizari



# Intrebari?

# Intrebari?