

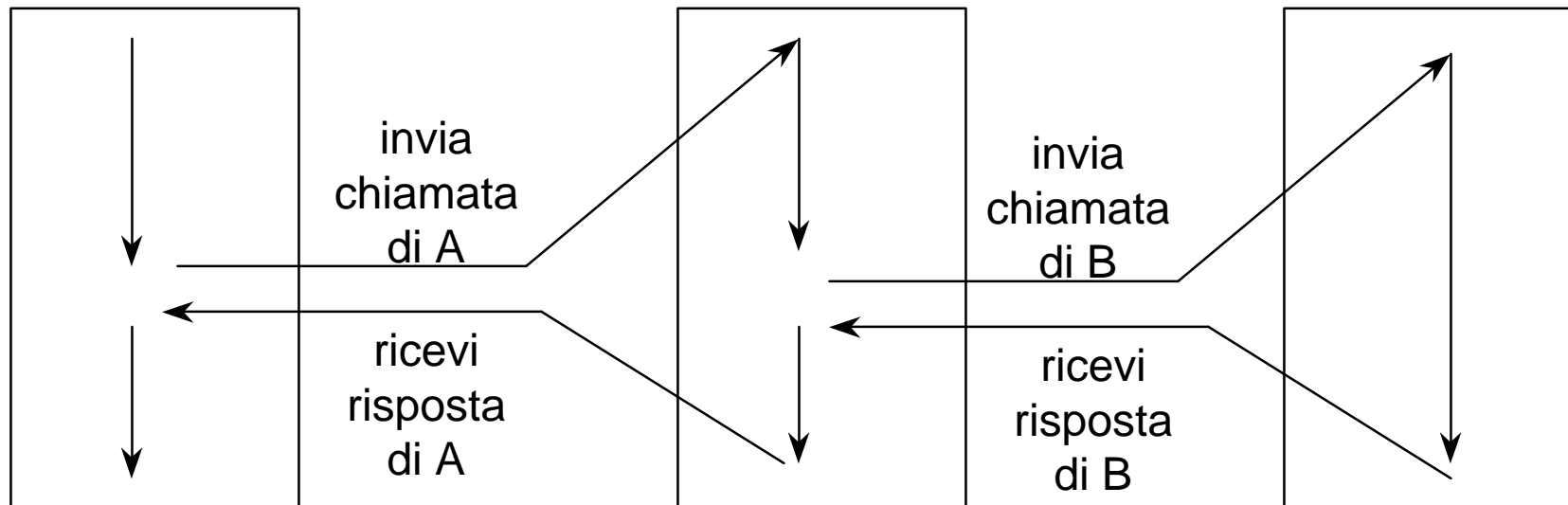
# Remote Procedure Call (RPC)

- È la trasposizione del meccanismo di chiamata di procedura (locale) ad un ambiente **distribuito**:

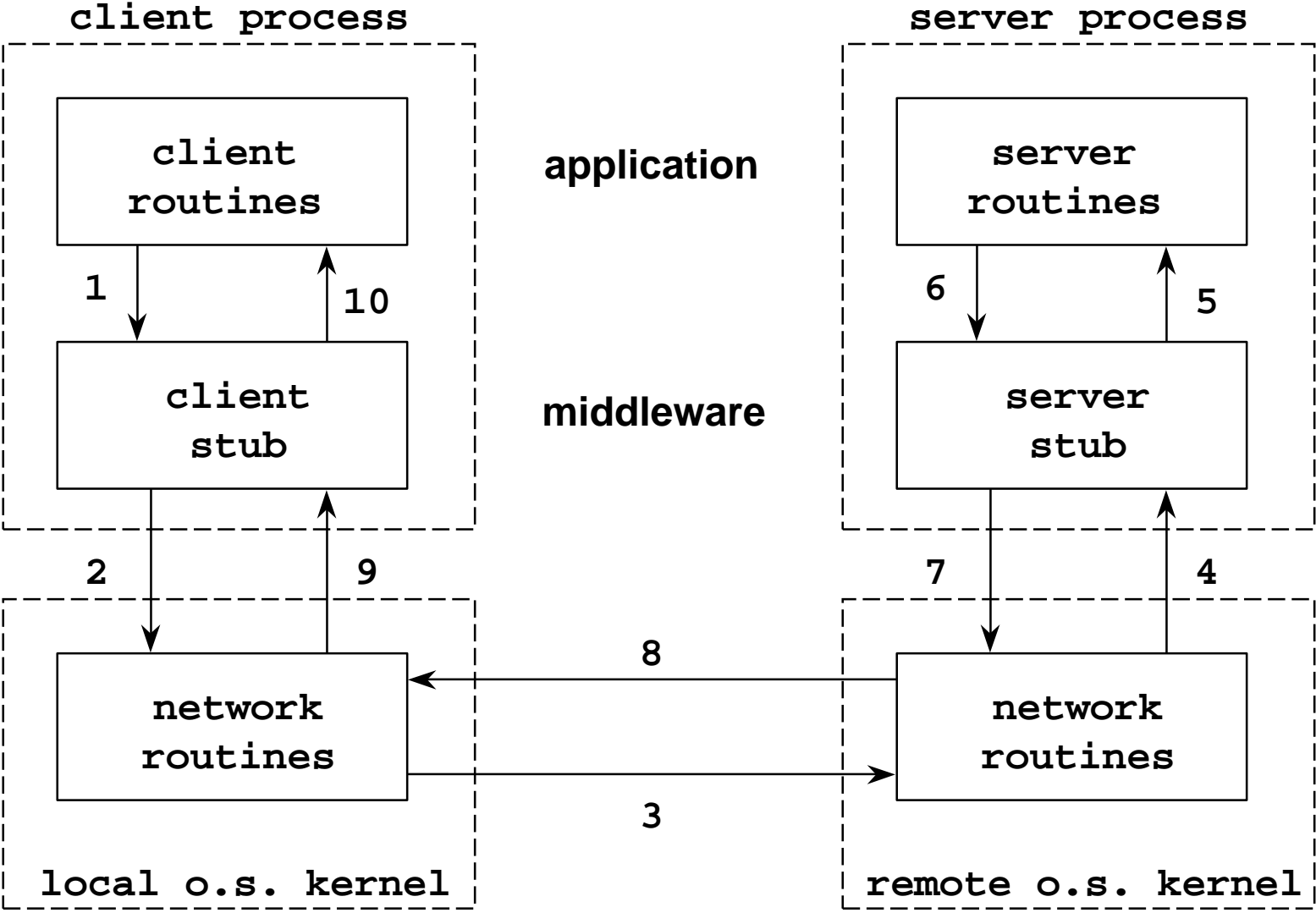
main su host 1

procedura A su host 2

procedura B su host 3



# Il modello RPC



# Particolarità delle Procedure Remote

- Chiamante e chiamato girano su processi diversi
  - servono meccanismi di *localizzazione* del server, prima della chiamata (linking dinamico)
  - occorre *sincronizzare* chiamate multiple in arrivo sul server
  - in caso di loop infinito o crash parziale, occorrono meccanismi di *timeout e recovery*
- Chiamante e chiamato hanno spazi di indirizzi disgiunti
  - Il passaggio dei parametri per riferimento non viene normalmente consentito

# Particolarità delle Procedure Remote

- Chiamante e chiamato sono eterogenei (diversi linguaggi, piattaforme hw/sw, ecc.)
  - parametri e valore di ritorno devono essere convertiti (*Marshaling*)
- Devono essere trattati i problemi di *security* (in particolare quelli di autenticazione)
- Una chiamata remota è più lenta di una chiamata locale (di alcune volte)

# Fallimenti e semantica delle chiamate remote

- Una chiamata remota può fallire (crash del server, crash o inaffidabilità della rete) o essere eseguita più volte (duplicazione dei messaggi)
  - ⇒ il middleware deve gestire queste situazioni
- A seconda del protocollo RPC usato e dello stato finale, possono essere garantite diverse semantiche:
  - **exactly once** : la procedura è stata eseguita esattamente una volta (protocollo sofisticato, incide sui tempi di esecuzione).
  - **at most once** : la procedura non è stata eseguita più di una volta (numeri di sequenza delle chiamate).
  - **at least once** : la procedura è stata eseguita una o più volte (timeout e ritrasmissioni).

# Esempi di RPC

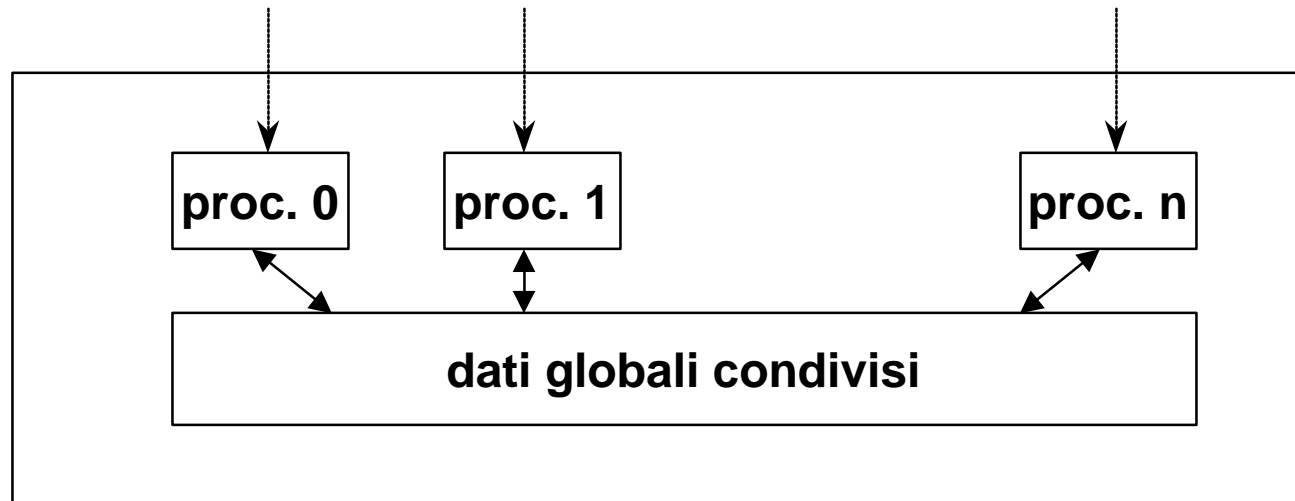
- SUN Microsystems ONC (molto diffuso in ambiente Unix)
- OSF DCE RPC
- Microsoft RPC

# SUN RPC (ONC)

- E' il meccanismo RPC più noto e diffuso
- *trasporto*: rete TCP/IP (può utilizzare TCP oppure UDP)
- *rappresentazione dei dati*: **XDR**
- *generazione automatica degli stub* (**rpcgen**)
  
- Documentazione:
  - Rfc 1057 (Protocollo RPC)
  - Rfc 1014 (linguaggio XDR)

# Programmi RPC

- Un **Programma RPC** è
  - un insieme di procedure accessibili tramite un server
  - che operano su dati condivisi



# Identificazione di Programmi e Procedure

- Ogni programma RPC viene identificato univocamente tramite un intero su 4 byte:
  - 0x00000000 - 0x1fffffff assegnati dalla SUN
  - 0x20000000 - 0x3fffffff definibili dall'utente
  - 0x40000000 - 0x5fffffff per uso temporaneo
  - 0x60000000 - 0xffffffff riservati per usi futuri
- Un programma può essere disponibile in più **versioni** (numerate a partire da 1)
- Per ogni versione è disponibile un diverso insieme di **procedure** (anche queste numerate)
- procedura 0 : è sempre la “*procedura di test*”

# Identificazione di Programmi e Procedure

- La procedura da chiamare viene dunque identificata tramite la terna: *(programma, versione, procedura)*
- È inoltre necessario identificare, tra tutti i server che rendono disponibile la procedura richiesta, quello su cui la si vuole eseguire.

# Esempi di numeri di programma pre-assegnati da SUN

port mapper	100000
rstat	100001
remote users	100002
nfs	100003
yp (NIS)	100004
mount	100005
DBX	100006
yp binder	100007
rwall	100008
yppasswd	100009
ethernet statistics	100010

# Parametri e valore di ritorno

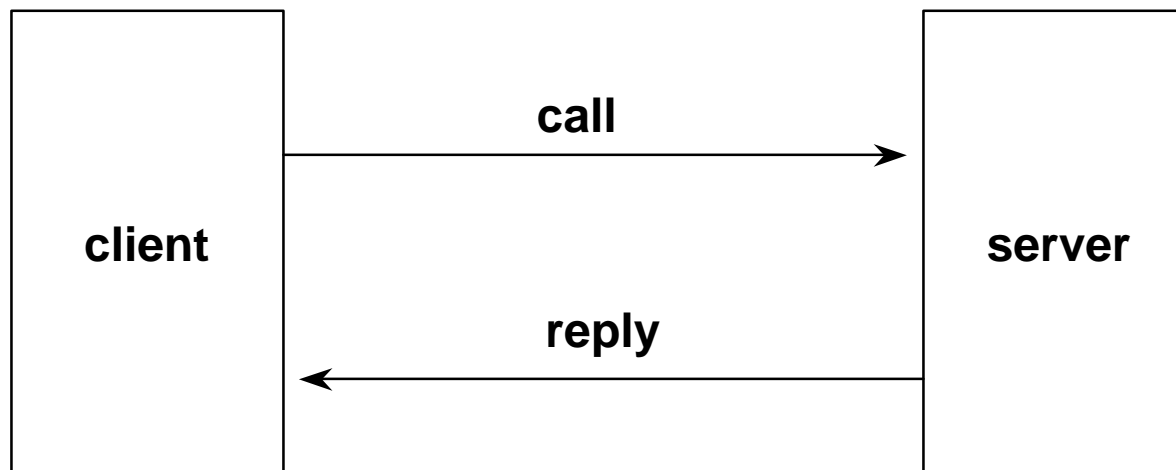
- Vengono rappresentati e trasferiti in formato XDR
- Per semplicità viene ammesso un unico parametro.
  - se si vogliono più parametri occorre definirli come campi di una struttura

# Autenticazione

- Il protocollo prevede la possibilità di scegliere tra diversi meccanismi di autenticazione:
  - nessuna autenticazione
  - autenticazione UNIX-like
  - autenticazione basata su protocollo crittografico (DES, Kerberos, ecc.)

# I Messaggi del Protocollo RPC

- Il protocollo prevede una chiamata (call) ed una risposta (reply):



# Struttura del Messaggio (in XDR)

```
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};
```

```
enum msg_type {
    CALL = 0,
    REPLY = 1
};
```

# Corpo della Chiamata

```
struct call_body {
    unsigned int rpcvers; /* must be equal to 2 */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure specific parameters start here */
};
```

# Campo di Autenticazione

```
enum auth_flavor {
    AUTH_NULL          = 0,
    AUTH_UNIX          = 1,
    AUTH_SHORT          = 2,
    AUTH_DES            = 3
    /* and more to be defined */
};
```

```
struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

# Corpo della Risposta

```
union reply_body switch (reply_stat stat) {  
    case MSG_ACCEPTED:  
        accepted_reply areply;  
    case MSG_DENIED:  
        rejected_reply rreply;  
} reply;
```

```
enum reply_stat {  
    MSG_ACCEPTED = 0,  
    MSG_DENIED   = 1  
};
```

# Chiamata accettata

```
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
    case SUCCESS:
        opaque results[0];
        /*procedure-specific results start here
        */
    case PROG_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    default:
        /* Void.  Cases include PROG_UNAVAIL,
        PROC_UNAVAIL, and GARBAGE_ARGS.
        */
        void;
    } reply_data;
};
```

```
enum accept_stat {
    SUCCESS          = 0, /* RPC executed successfully      */
    PROG_UNAVAIL     = 1, /* remote hasn't exported program */
    PROG_MISMATCH    = 2, /* remote can't support version # */
    PROC_UNAVAIL     = 3, /* program can't support procedure */
    GARBAGE_ARGS     = 4  /* procedure can't decode params  */
};
```

# Chiamata rifiutata

```
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};

enum reject_stat {
    RPC_MISMATCH=0, /* RPC version number != 2          */
    AUTH_ERROR=1   /* remote can't authenticate caller */
};
```

# Semantica delle chiamate

- Il protocollo Sun RPC non prevede meccanismi per garantire l'affidabilità: essa dipende dal trasporto usato
  - Il middleware RPC implementa una semplice politica di ritrasmissione
    - con timeout e numero di ritrasmissioni non adattativi (l'utente può impostarli per una data applicazione)
    - raggiunto il numero massimo di ritrasmissioni, la chiamata viene abortita (fallisce)
- NB: il fallimento non significa che la procedura non sia stata eseguita

# Semantica delle chiamate

- In pratica, l'applicazione
  - deve essere consapevole del trasporto usato
  - può trarre le minime conclusioni compatibili con le caratteristiche del trasporto.
- Esempio: usando il trasporto UDP:
  - se la chiamata ha successo: *at least once*
  - se la chiamata fallisce: *zero or more*
- Una strategia comunemente usata è quella di rendere le procedure **idempotenti**. Esempio:

appendi X al file Y   =>   scrivi X in posizione k nel file Y

# Concorrenza e Sincronizzazione

- La sincronizzazione viene ottenuta serializzando le chiamate:
  - ogni server non può eseguire più di una chiamata per volta

# Localizzazione del Server

- Per localizzare un server occorre individuarne IP address, protocollo(TCP/UDP) e numero di porta.
- Problema: i numeri di porta sono limitati
  - non è possibile un'assegnazione statica porta-programma
  - viene usato un mapping dinamico ed un **port-mapper** (server che gestisce il database delle corrispondenze porta-programma su un dato host)

# Port Mapper

- È un server RPC, presente su ogni host, che usa la porta riservata 111 (assegnata staticamente)
- Un server RPC che rende disponibile un programma deve *registrarsi* presso il port mapper locale, ottenendo da esso un numero di porta
- Un client RPC che voglia attivare una procedura remota,
  - interroga il port mapper dell'host per conoscere il numero di porta su cui è disponibile il programma desiderato
  - contatta il server al numero di porta così conosciuto

# Generazione degli Stub

- Gli stub possono essere generati
  - manualmente, facendo uso di una libreria in linguaggio C
  - automaticamente dal programma rpcgen
- L'input di rpcgen è un file di specifica, che descrive un programma RPC (la sua interfaccia) in un apposito linguaggio basato su XDR

# Utilizzo di rpcgen

client routines

`client_date.c`

RPC specification

`date.x`

server routines

`server_date.c`

rpcgen

client stub

`date_clnt.c`

server stub

`date_svc.c`

common header  
& XDR functions

`date.h`

`date_xdr.c`

cc

cc

client program

`client_date`

RPC library

server program

`server_date`

# Descrizione Formale dei Programmi RPC

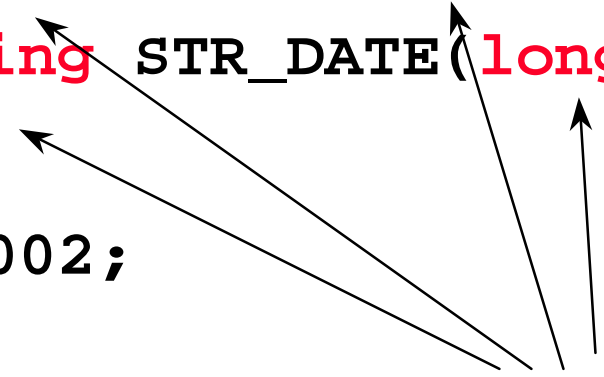
- Un file di specifica
  - ha sintassi C-like
  - contiene
    - » **le descrizioni XDR dei tipi di dato usati per parametri e valori di ritorno**
    - » **la descrizione dell'interfaccia di ogni versione del programma (prototipi delle funzioni offerte)**

# Esempio: Programma DATE

```
/* date.x */
```

```
program DATE_PROG {  
    version DATE_VERS {  
        long BIN_DATE(void) = 1;  
        string STR_DATE(long) = 2;  
    } = 1;  
} = 0x20000002;
```

*tipi XDR*



# Altro Esempio: il Programma Port Mapper

```
program PMAP_PROG {
  version PMAP_VERS {
    void          PMAPPROC_NULL(void)          = 0;

    bool          PMAPPROC_SET(mapping)        = 1;

    bool          PMAPPROC_UNSET(mapping)       = 2;

    unsigned int  PMAPPROC_GETPORT(mapping)    = 3;

    pmaplist      PMAPPROC_DUMP(void)         = 4;

    call_result   PMAPPROC_CALLIT(call_args)  = 5;
  } = 2;
} = 100000;
```

# Tipi XDR del Port Mapper

```
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};
```

```
struct *pmaplist {
    mapping map;
    pmaplist next;
};
```

```
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};
```

```
struct call_result {
    unsigned int port;
    opaque res<>;
};
```

# Trasformazione delle procedure operata da rpcgen

- Il **nome** della procedura viene modificato:
    - i caratteri originali diventano minuscoli
    - viene aggiunto il numero di versione preceduto da underscore
  - Parametro e valore di ritorno diventano *puntatori*
  - Viene aggiunto un nuovo parametro:
    - un “HANDLE” (cioè un riferimento) per lo stub del client
- NB: l'handle varia a seconda di quale *server*, *programma*, *versione*, *protocollo* vengono usati

# File header generato da rpcgen

```
/* date.h */
```

```
#ifndef _DATE_H_RPCGEN  
#define _DATE_H_RPCGEN
```

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

```
#define DATE_PROG ((u_long)0x20000002)  
#define DATE_VERS ((u_long)1)
```

```
#define BIN_DATE ((u_long)1)
```

```
extern long * bin_date_1(void *, CLIENT *);
```

```
extern long * bin_date_1_svc(void *, struct svc_req *);
```

```
#define STR_DATE ((u_long)2)
```

```
extern char ** str_date_1(long *, CLIENT *);
```

```
extern char ** str_date_1_svc(long *, struct svc_req *);
```

generaz. nome function:

- minuscolo
- aggiunge num. vers.

aggiunto parametro  
(handle cliente)

# Libreria RPC

- Fornisce una serie di funzioni per:
  - eseguire le chiamate RPC
  - registrare servizi sul port mapper
- Gli stub si appoggiano a queste funzioni per realizzare il meccanismo di chiamata in modo trasparente.
- È possibile intervenire sugli stub generati automaticamente per modificare certi parametri (p. es. numero massimo di tentativi).

# Libreria RPC: creazione e rilascio del client handle

```
CLIENT *clnt_create (server, pnum, vnum, prot)
char * server      nome dell'host server
int pnum          numero del programma
int vnum          numero di versione
char * prot       protocollo di trasporto usato (TCP o UDP)
```

- Inizializza lo stub e contatta il port mapper del server
- Restituisce l'HANDLE allo stub inizializzato

```
void clnt_destroy (handle)
CLIENT * handlehandle da rilasciare
```

# Libreria RPC: interazione con il port mapper (I)

```
struct pmaplist *pmap_getmaps (addr)  
struct sockaddr_in * addr indirizzo dell'host server
```

- Restituisce la lista delle assegnazioni di porta

```
int pmap_getport (addr, pnum, vnum, prot)  
struct sockaddr_in * addr indirizzo dell'host server  
int pnum          numero del programma  
int vnum          numero di versione  
char * prot       protocollo di trasporto usato (TCP o UDP)
```

- Restituisce il numero di porta associato ai parametri dati

# Libreria RPC: interazione con il port mapper (II)

```
int pmap_set (pnum, vnum, prot, port)
int pnum      numero del programma
int vnum      numero di versione
char * prot   protocollo di trasporto usato (TCP o UDP)
int port      porta da assegnare
```

- Assegna il numero di porta dato alla terna data
- Restituisce 1 se l'assegnazione riesce

```
int pmap_unset (pnum, vnum)
```

- Elimina un'assegnazione

# Esempio di Client routines

```
/* client_date.c */

#include <stdio.h>
#include <rpc/rpc.h>
#include "date.h"

void main (int argc, char *argv[])
{
    char *server;
    CLIENT *cl;
    long *lresult;
    char **sresult;

    server = argv[1]; /* hostname: dato sulla linea di comando */

    cl = clnt_create(server, DATE_PROG, DATE_VERS, "tcp");

    lresult = bin_date_1(NULL, cl);          /* chiamata prima procedura */
    printf("tempo su host %s = %ld\n", server, *lresult);

    sresult = str_date_1(lresult, cl);      /* chiamata seconda procedura */
    printf ("tempo su host %s = %s\n", server, *sresult);

    clnt_destroy(cl);                       /* distruzione del client */
}
```

# Esempio di Server routines

```
/* server_date.c */

#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/time.h>
#include "date.h"

long * bin_date_1()
{
    static long timeval;

    time(&timeval);
    return (&timeval);
}

char ** str_date_1(long *bintime)
{
    static char *ptr;

    ptr = ctime(bintime);
    return(&ptr);
}
```

# Esempio di Client Stub

```
/* server_date.c */

#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/time.h>
#include "date.h"

long * bin_date_1()
{
    static long timeval;

    time(&timeval);
    return (&timeval);
}

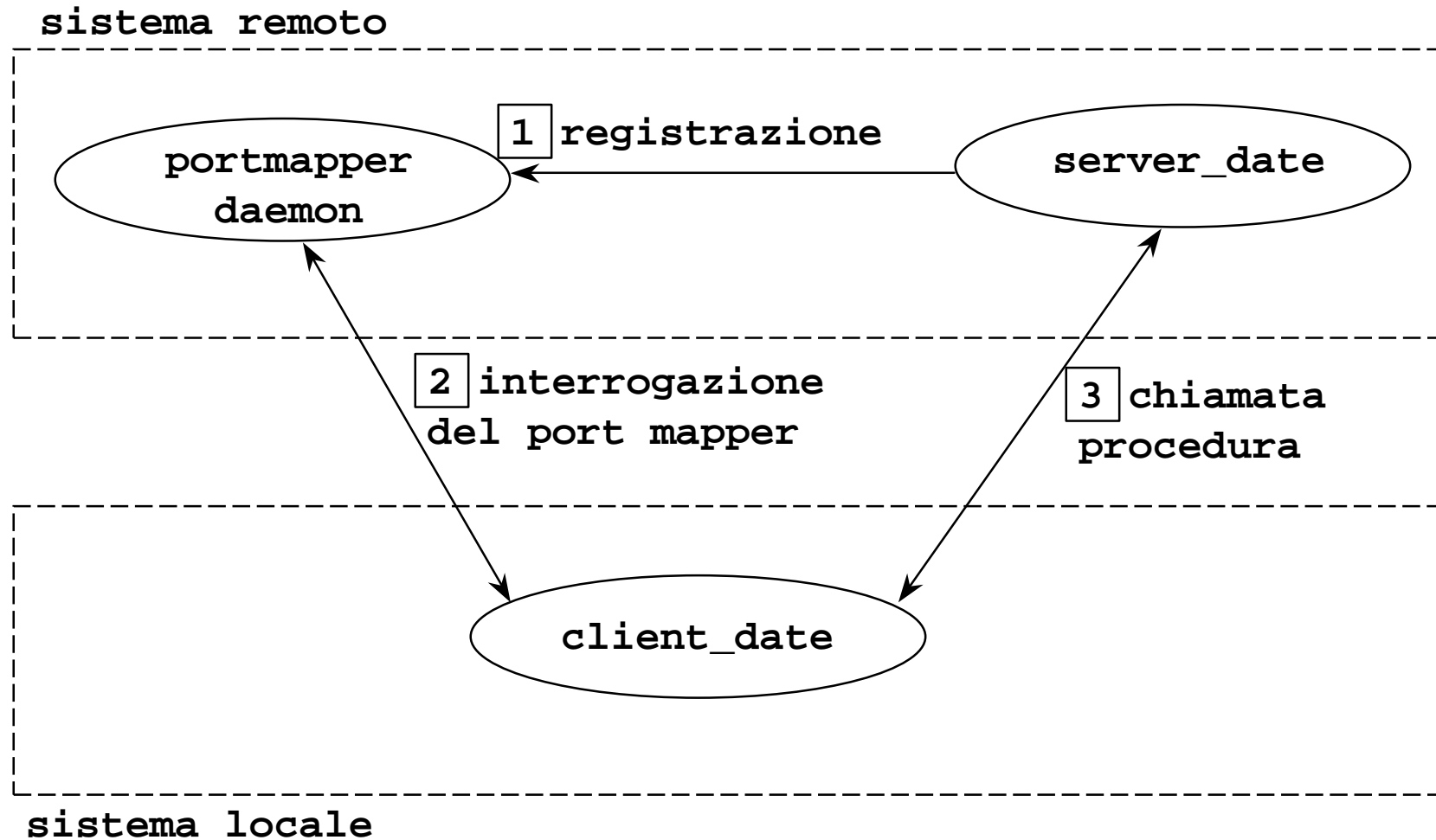
char ** str_date_1(long *bintime)
{
    static char *ptr;

    ptr = ctime(bintime);
    return(&ptr);
}
```

# Esempio di Client Stub

# Esempio di Server Stub

# Funzionamento



# Sviluppo di Applicazioni Distribuite con i Socket

- I socket forniscono solo le funzioni di livello 4 (trasporto)
- Il programmatore deve implementare un opportuno protocollo di interazione, realizzando:
  - una codifica dei dati scambiati
  - delle procedure di interazione
- L'enfasi è normalmente più sulla comunicazione che sull'applicazione

# Sviluppo di Applicazioni Distribuite con RPC

- Si sviluppa e si testa un'applicazione centralizzata
- Si dividono i moduli dell'applicazione su più calcolatori, facendoli comunicare con RPC.
- Vantaggi:
  - il programmatore può concentrarsi sull'applicazione anziché sulla comunicazione

# Esempio

- Rubrica telefonica