

Sistemi Operativi II

Corso di Laurea in Ingegneria Informatica

Facolta' di Ingegneria, Universita' "La Sapienza"

Docente: Francesco Quaglia

## **Architetture client/server:**

1. Nozioni di base
2. RPC

# Paradigma client/server

- nato a cavallo tra la fine degli anni '80 e l'inizio degli anni '90
- paradigma di comunicazione interprocesso piu' complesso delle semplici primitive di scambio messaggi
- i processi applicativi vengono suddivisi in fruitori di servizi (processi client) e fornitori di servizi (processi server)
- una interazione tra client e server si svolge nel modo seguente:
  1. un client invia una richiesta di servizio al server
  2. il server esegue il servizio ed infine invia una replica al client
- più server possono fornire lo stesso servizio
- un servizio può essere implementato attraverso la cooperazione di più server
- un server può anche comportarsi come un client fruendo di servizi di altri server
- l'interazione è, almeno teoricamente, indipendente dal tipo di piattaforma (sistema operativo, meccanismi base di comunicazione)

# Interazione client/server

- un client deve impacchettare la sua richiesta in modo tale che questa venga compresa dal server
- il server riceve la richiesta, esegue il servizio e reinvia la risposta in un modo comprensibile dal client
- i passi sono gli stessi di una applicazione che effettui una chiamata ad una procedura locale definita in qualche libreria
- vedere un'interazione client/server come un qualcosa di simile ad una chiamata a procedura locale:
  1. impone al server una interfaccia standard per il servizio
  2. fa sì che il programmatore possa invocare un servizio remoto in modo semplice
- come nelle procedure locali, un servizio può essere reso più efficiente in modo trasparente al client (a patto che l'interfaccia rimanga inalterata)

# Aspetti peculiari

- la perdita di un messaggio di richiesta o di replica può condurre alla non esecuzione o alla esecuzione molteplice dello stesso servizio
- vi è quindi necessità di vincoli semantici sull'esecuzione di una interazione client/server
- client e server non vedono lo stesso spazio di indirizzamento (effetto sul passaggio dei parametri)
- la rappresentazione delle informazioni può essere disomogenea ai due end-point

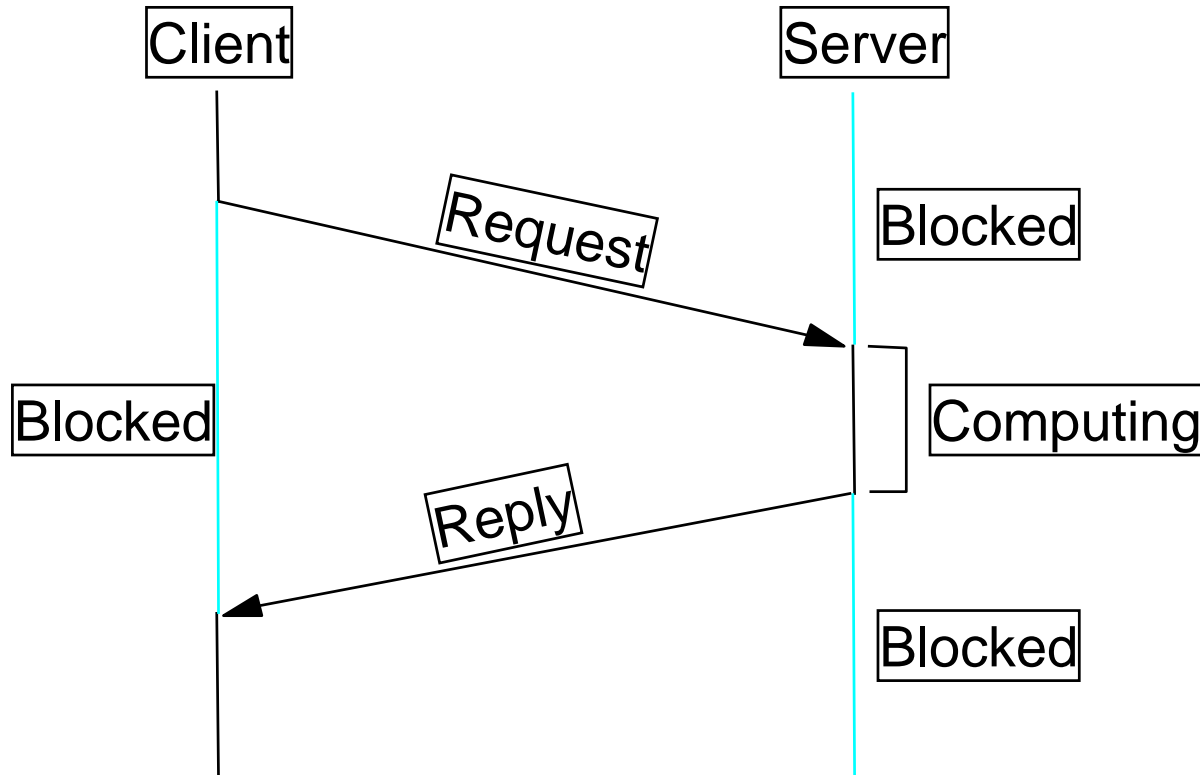
---

Remote Procedure Call (RPC), o Chiamata a Procedura Remota, è una tecnica risolutiva per l'interazione Client/Server

# Identificazione per l'RPC

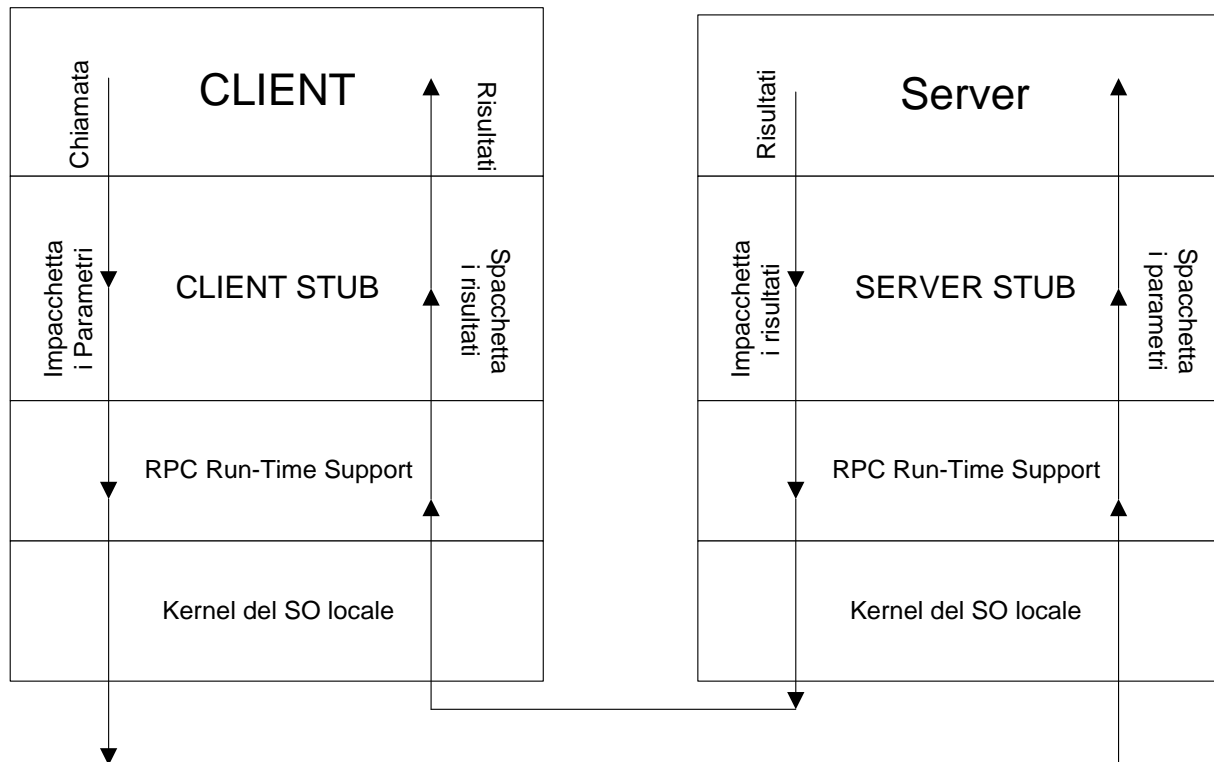
- **A tempo di scrittura del codice.** Le RPC usate/fornite dovranno essere dichiarate esplicitamente dal programmatore attraverso import/export delle definizioni delle interfacce.
- **A tempo di esecuzione.** Ogni macchina su cui è in esecuzione un programma client e/o server dovrà avere un supporto a tempo si esecuzione per le RPC (RPC run-time support) in grado di eseguire alcune operazioni delle RPC come ad esempio la localizzazione del server o la registrazione di un nuovo servizio offerto da un nuovo server.
- **A tempo di compilazione.** Durante la compilazione per ogni chiamata a procedura remota vengono agganciate linee di codice al programma originario (stub) che permettono operazioni standard sui dati (impacchettamento e codifica universalmente riconosciuta) e le chiamate al RPC run-time support.
- **A tempo di collegamento.** Durante il collegamento il programma sorgente viene collegato al RPC run-time support per ottenere il codice eseguibile.

# RPC timeline



# Meccanismi per RPC

- un meccanismo che nasconda le insidie della rete (perdita di pacchetti e riordinamento dei messaggi) e della separazione client/server a livello di sistema
- un meccanismo per impacchettare gli argomenti dal lato chiamante e per spaccettarli dal lato chiamato



# Localizzazione del server

## Metodo Statico

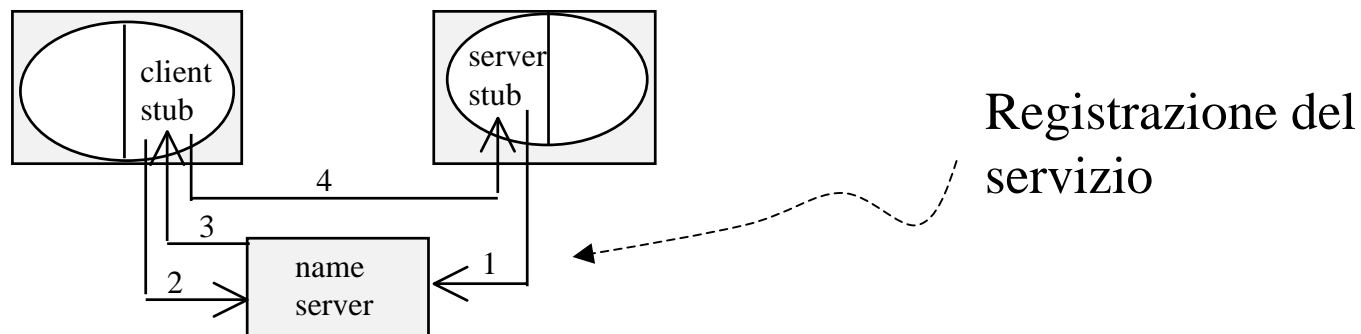
- cablare all'interno del client l'indirizzo (IP address) del server

## Metodo Dinamico

- lo stub del client invia un broadcast richiedendo l'indirizzo di un server in grado di eseguire la RPC desiderata
- il supporto run-time delle RPC di ogni macchina risponde se il servizio richiesto e' fornito da un suo server in esecuzione

## Name Server

- il client alla ricerca di un server consulta una entità, name server, la quale gestisce una lista di associazioni server-servizi



# Passaggio dei Parametri

- **Call by Reference – inapplicabile**
- **Call by Copy/Restore.** Copia una variabile  $a$ , da parte dello stub del client, nel pacchetto dati (come se fosse passata per valore). Il nuovo valore di  $a$ , restituito dal server nei parametri di ritorno della RPC sarà copiato, dallo stub del client, nella cella di memoria relativa alla variabile  $a$ .

## CLIENT SIDE

begin

.....

$a=0$ ;

*doppioincr(a,a)*;

*writeln (a)*; ...

end

## SERVER SIDE

procedure *doppioincr* (var  $x,y$ : integer)

begin

.....

$x:= x+2$ ;

$y:= y+3$ ;

end

*Risultato: Call by ref,  $a=“5”$*

*Call by copy/restore  $a= “2”$  o “3”*

*dipendente dall’implementazione dello stub del client*

# Stub compilers

- the compiler must know which parameters are *in* parameters and which are *out*
  - in parameters are sent from the client to server, out parameters are sent back
  - languages like C have no concept of *in* or *out* parameters
  - therefore the compiler cannot be a standard C compiler, and the specification of the procedures cannot be done in standard C language
- 

A typical specification might be: `int max(in int x, in int y, out int z);`

A stub compiler would use this to generate the two stubs

# Semantica delle RPC

## *“At least once”*

- servizio eseguito almeno una volta per invocazione
- meccanismi:
  - time-out stub del client
  - ritrasmissione

## *“At most once”*

- servizio eseguito al più una volta per invocazione
- meccanismi
  - codice di errore di ritorno (non c'è garanzia sul processamento)

## *“Exactly once”*

- servizio eseguito esattamente una volta per invocazione

# Meccanismi per exactly-once

## Lato server

- immagazzinare tutti i risultati delle RPC nel server (logging)
- se arriva al server una richiesta già effettuata il risultato dovrà essere preso dal file di log

## Lato Client

- numerare tutte le richieste dai client (sequence number)
- numero di reincarnazione (add 1 ad ogni restart del client)
- a seguito di un guasto un client invia il numero di reincarnazione corrente prima di cominciare ad eseguire le RPC (per uccidere le RPC pending della incarnazione precedente)

# Sottosistema di comunicazione

TCP – troppo costoso in fase di connessione

UDP – nessun costo di connessione ma si deve gestire al di sopra un protocollo per l'invio affidabile dei dati

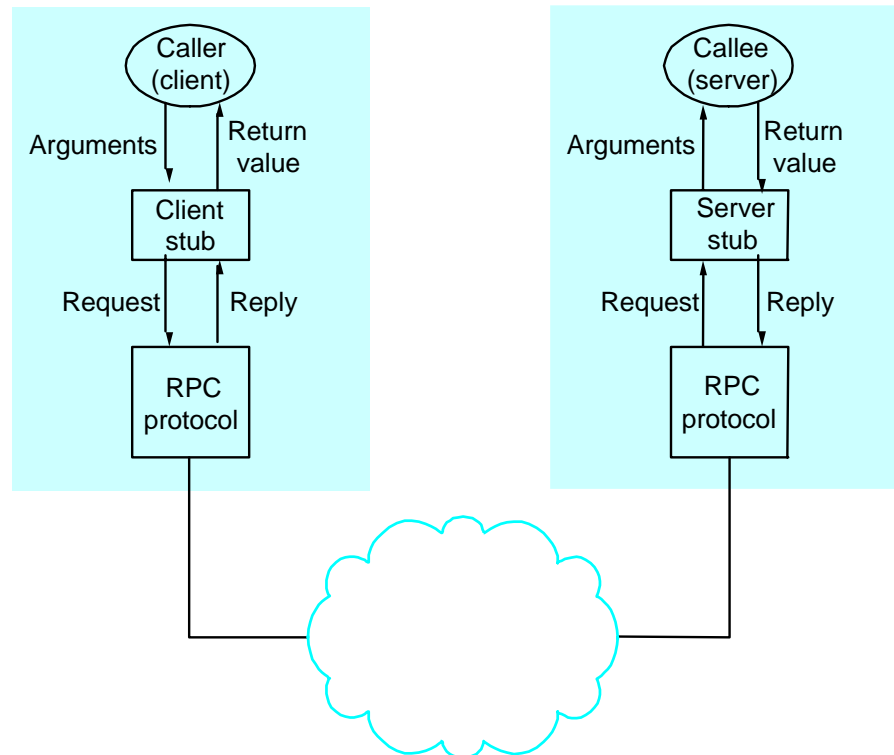
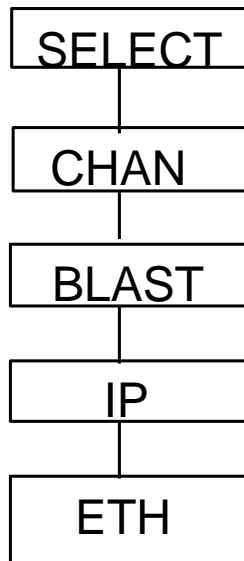
IP – dobbiamo gestire anche il multiplexing/demultiplexing dei pacchetti all'interno del singolo host oltre ai problemi che derivano dall'utilizzo di UDP

## Gestione di pacchetti di riscontro

- Stop and wait
- Blast (tutti i pacchetti sono inviati in sequenza ed il server invia un ack in ricezione dell'ultimo pacchetto)

# Un esempio di componenti RPC

- protocol Stack
  - BLAST: **fragments** and reassembles large messages
  - CHAN: synchronizes request and reply **messages** (at most once semantic)
  - SELECT: dispatches request to the **correct process**
- stubs

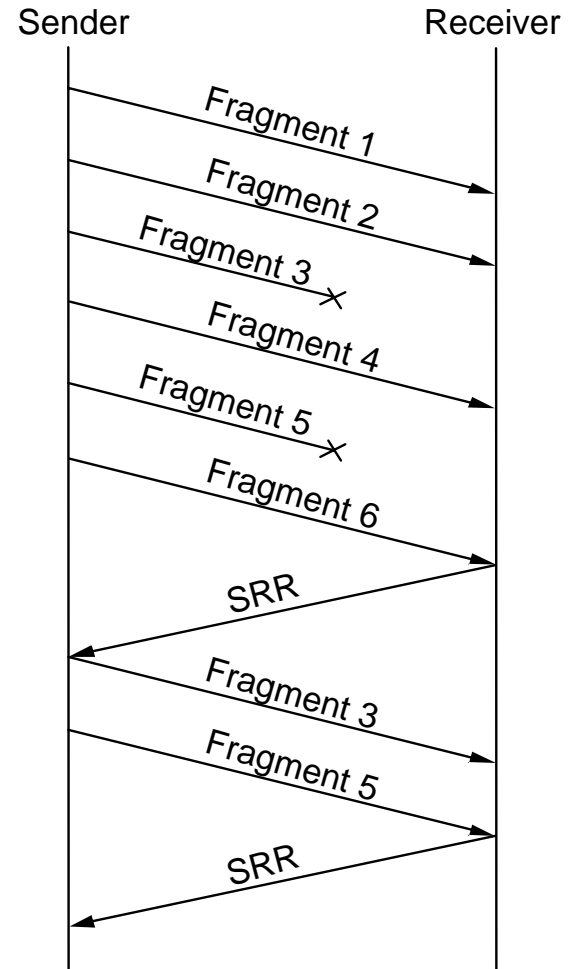


Similar to SunRPC

# Bulk-transfer: protocollo BLAST

## Strategia

- Ritrasmissione selettiva
- Acknowledgment parziale
- Uso di timer
  - DONE
  - LAST\_FRAG
  - RETRY



# Dettagli sul protocollo BLAST

## Lato sender

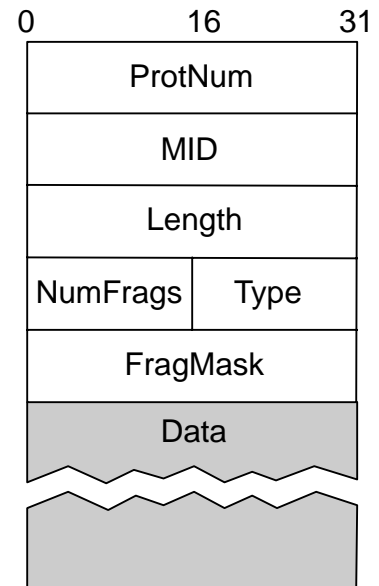
- store fragments in local memory, send all fragments, set timer DONE
- if receive SRR, send missing fragments and re-set timer DONE
- if receive SRR “all fragments have been received”, then sender frees fragments
- if timer DONE expires, free fragments (sender gives up)

## Lato receiver

- when first fragment arrives, set timer LAST\_FRAG
- when all fragments present, reassemble and pass up
- four exceptional conditions:
  - if last fragment arrives but message not complete send SRR and set timer RETRY
  - if timer LAST\_FRAG expires: send SRR and set timer RETRY
  - if timer RETRY expires for first or second time: send SRR and set timer RETRY
  - if timer RETRY expires a third time: give up and free partial message
- **problem: what's happen if all fragments of a message are missings?**

# Formato dell'header BLAST

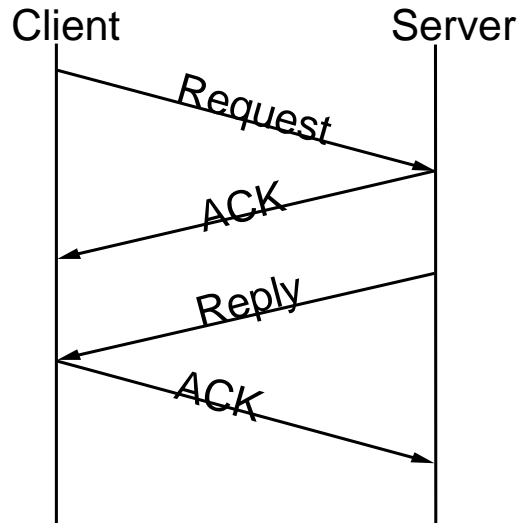
- MID (Message ID) must protect against wrap around (all fragments of a message have the same MID)
- TYPE = DATA or SRR
- NumFrag indicates number of fragments
- FragMask distinguishes among fragments
  - if Type=DATA, identifies this fragment
  - if Type=SRR, identifies missing fragments
  - Max 32 fragments per message



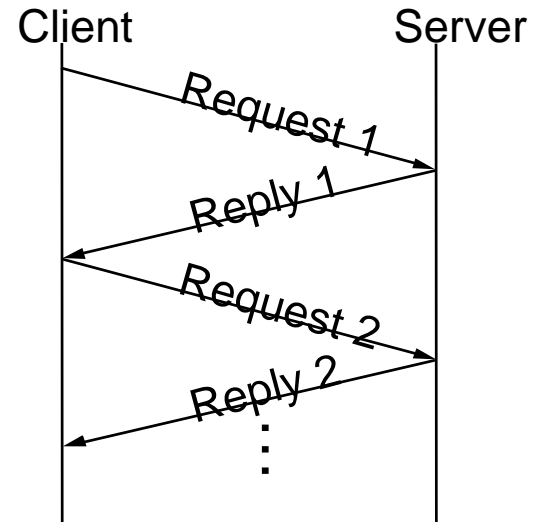
# Protocollo CHAN: Request/Reply

- garantisce la consegna dei messaggi
- sincronizza client e server
- supporta la semantica **at-most-once**

## Simple case



## Implicit Acks



# Dettagli

- Lost message (request, reply, or ACK)
  - set RETRANSMIT timer
  - use message id (MID) field to distinguish
- Slow (long running) server
  - client periodically sends “are you alive” probe, or
  - server periodically sends “I’m alive” notice
- Want to support multiple outstanding calls
  - use channel id (CID) field to distinguish
- Machines crash and reboot
  - use boot id (BID) field to distinguish
- Use RETRANSMIT (client), RETRANSMIT (Server) and PROBE (Client)

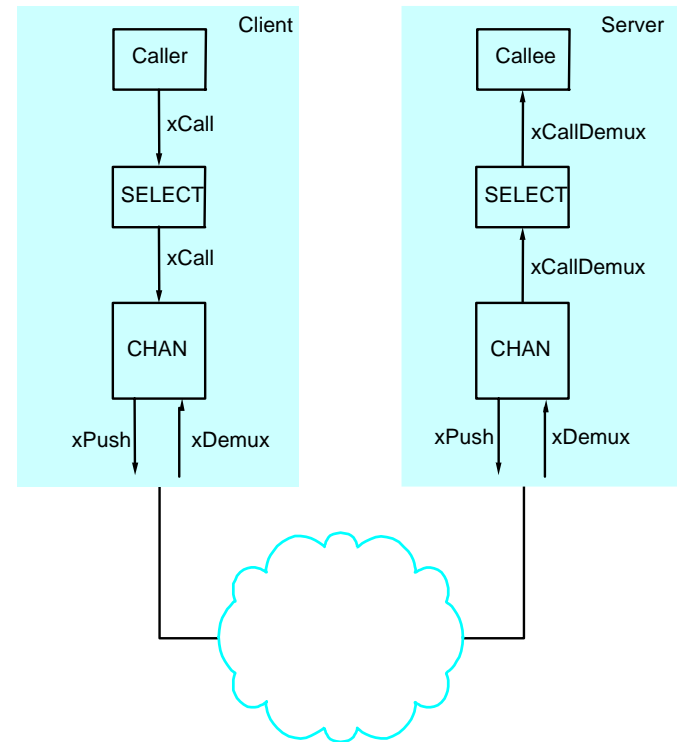
# Formato dell'header CHAN

```
typedef struct {
    u_short  Type;      /* REQ, REP, ACK, PROBE */
    u_short  CID;      /* unique channel id */
    int      MID;      /* unique message id */
    int      BID;      /* unique boot id */
    int      Length;   /* length of message */
    int      ProtNum;  /* high-level protocol */
} ChanHdr;
```

```
typedef struct {
    u_char   type;      /* CLIENT or SERVER */
    u_char   status;    /* BUSY or IDLE */
    int      retries;   /* number of retries */
    int      timeout;   /* timeout value */
    XkReturn ret_val;   /* return value */
    Msg      *request;  /* request message */
    Msg      *reply;    /* reply message */
    Semaphore reply_sem; /* client semaphore */
    int      mid;       /* message id */
    int      bid;       /* boot id */
} ChanState;
```

# Dispatcher (SELECT)

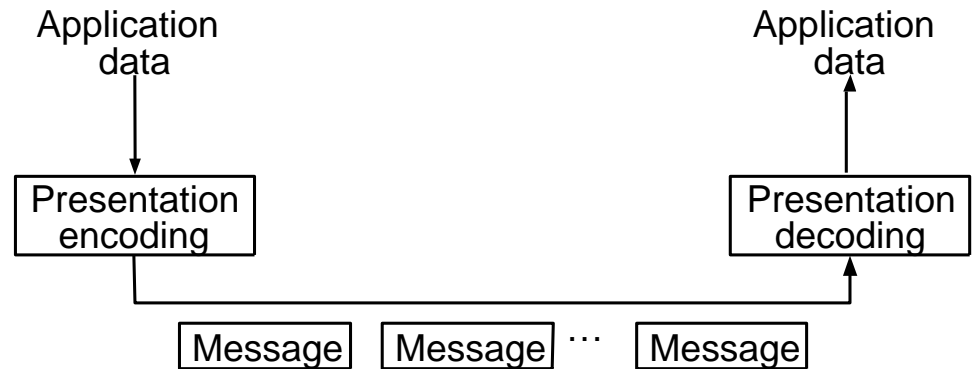
- Dispatch to appropriate procedure
- Counterpart to UDP



- Address Space for Procedures
  - flat: unique id for each possible procedure
  - hierarchical: program + procedure number

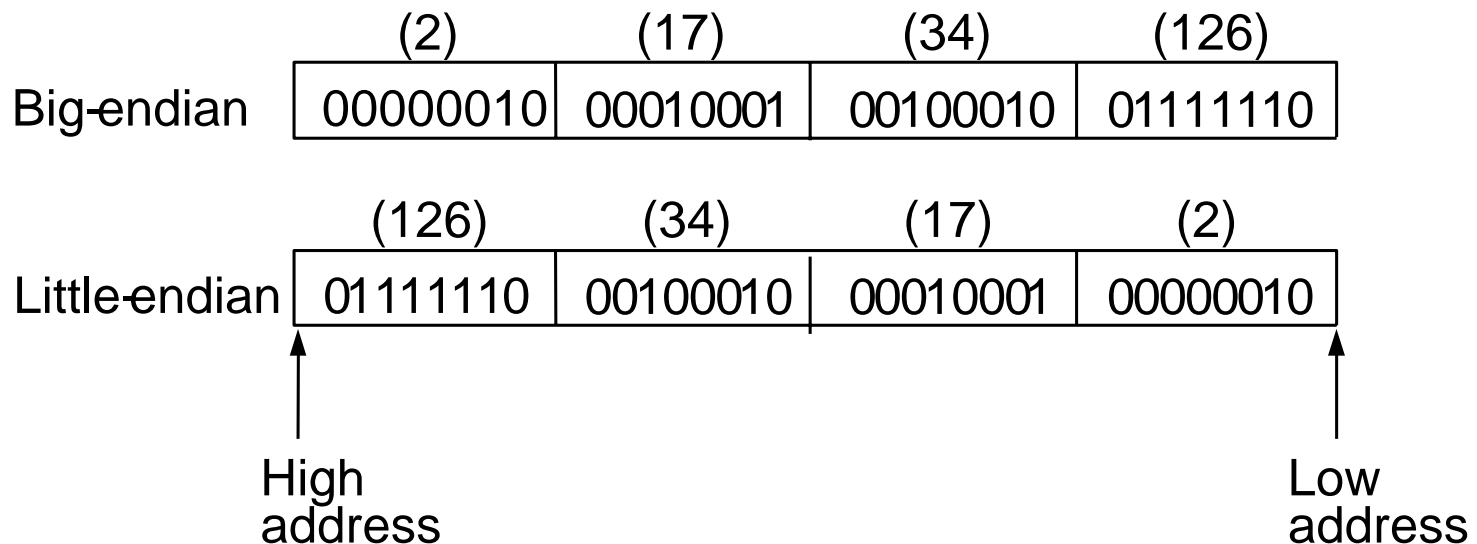
# Formato dei dati

- Marshalling  
(encoding)  
application data into  
messages
- Unmarshalling  
(decoding) messages  
into application data
- Data types  
considered
  - integers
  - floats
  - strings
  - arrays
  - structs



# Difficoltà

- representation of base types
  - floating point: IEEE 754 versus non-standard
  - integer: big-endian versus little-endian

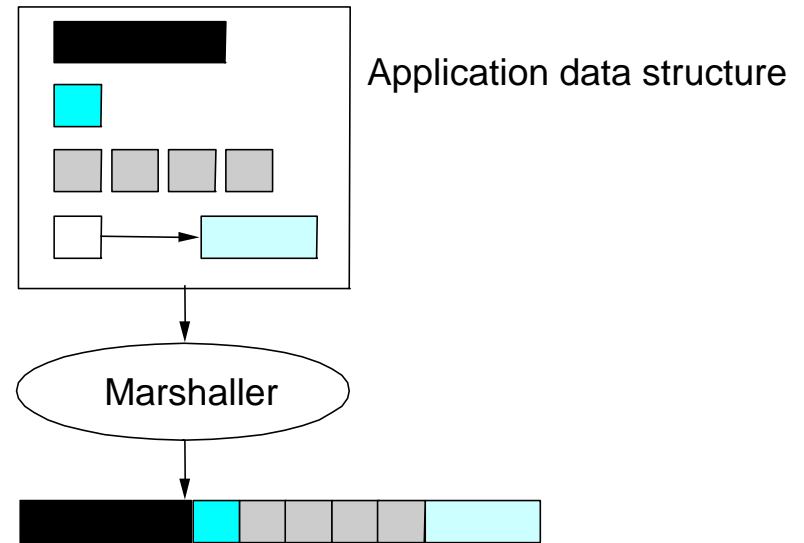


- different representation of integers (1,2,4 bytes)

# Taxonomy

## Data types

- base types (e.g., ints, floats); must convert
- flat types (e.g., structures, arrays); must pack
- complex types (e.g., pointers); must linearize



## Conversion Strategy

- Asymmetric
- Symmetric

# Asymmetric conversion

- Convert at one end (client or server)
- Must know the platform type of destination or source
- With  $N$  types need  $N(N-1)$  converters total
- Sometimes known as receiver-makes-right
- Basis for NDR (Network Data Representation)

# Symmetric conversion

- Convert to and from a **canonical intermediate form** (an external data representation)
- Flexible and portable, but at a cost in computation
- With  $N$  types need  $2N$  converters total
- Basis for XDR (eXternal Data Representation) and ASN.1 (Abstract Syntax Notation One)

# Data type specification

How a receiver knows which type of data is in the packet?

- Tagged data

type = INT	len = 4		value =	417892	
---------------	---------	--	---------	--------	--

- Type, len, architecture

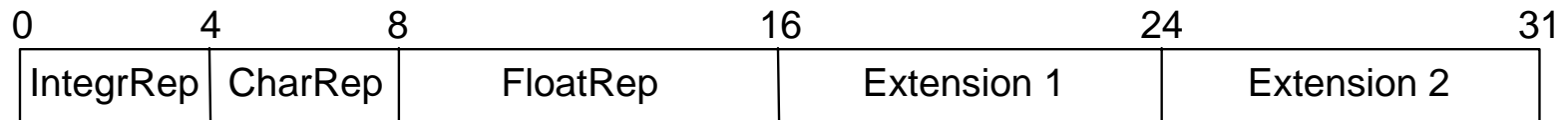
- Untagged data

- No variable size data structures

- End-to-end presentation formatting

# Network Data Representation (NDR)

- Defined by DCE (Distributed Computing Environment)
  - Essentially the C type system
  - Receiver-makes-right (architecture tag)
  - Individual data items untagged
  - Compiled stubs from IDL (Interface Definition Language)
  - 4-byte architecture tag
- IntegerRep
    - 0 = big-endian
    - 1 = little-endian
  - CharRep
    - 0 = ASCII
    - 1 = EBCDIC
  - FloatRep
    - 0 = IEEE 754
    - 1 = VAX
    - 2 = Cray
    - 3 = IBM

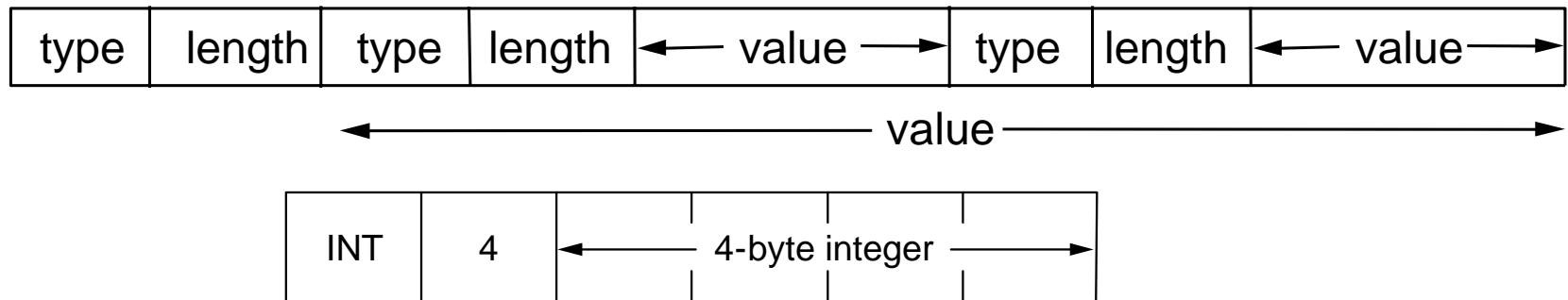


# Abstract Syntax Notation One (ASN-1)

- An ISO standard
- Essentially the C type system
- Canonical intermediate form
- Tagged
- BER: Basic Encoding Rules

(tag, length, value)

- Nested representation of data structures



# eXternal Data Representation (XDR)

- Defined by Sun for use with SunRPC
- C type system (without function pointers)
- Canonical intermediate form
- Untagged (except array length) – Coding data, not their type
- Type of data must be determined at application level
- Compiled stubs

# An example

```
struct example {  
    int count;  
    int values[2];  
    char buffer[4];  
}
```

count	values[2]	buffer [4]
6	2 450 498	4 ABCD

# Creating an XDR Data Stream

## Create buffer

```
xdrmem_create(xdrs, buf, BUFSIZE, XDR_ENCODE)
```

## Make calls to build buffer

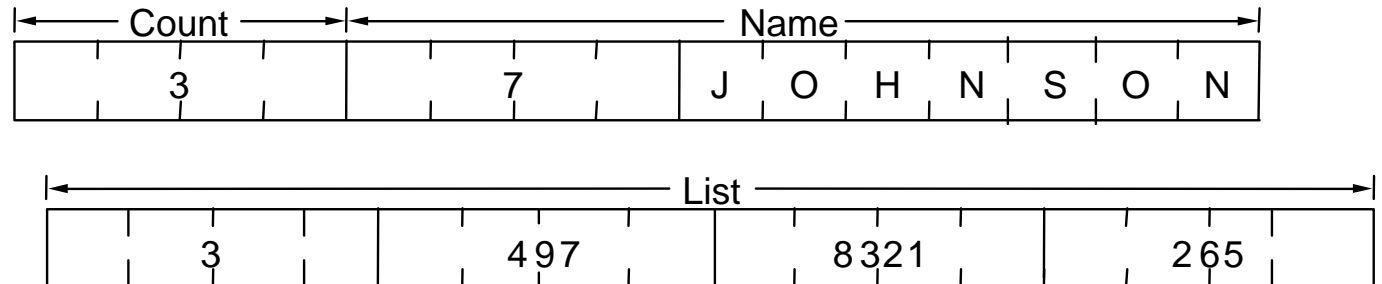
```
int i = 300;  
xdr_int(xdrs, &i);
```

# XDR library routines

```
#define MAXNAME 256;  
#define MAXLIST 100;
```

```
struct item {  
    int     count;  
    char    name[MAXNAME];  
    int     list[MAXLIST];  
};
```

```
bool_t  
xdr_item(XDR *xdrs, struct item *ptr)  
{  
    return(xdr_int(xdrs, &ptr->count) &&  
           xdr_string(xdrs, &ptr->name, MAXNAME) &&  
           xdr_array(xdrs, &ptr->list, &ptr->count,  
                     MAXLIST, sizeof(int), xdr_int));  
}
```



Valid data types supported by XDR include

- int
- unsigned int
- long
- structure
- fixed array
- string (null terminated char \*)

## RPC specification

A file with a ``.x" suffix acts as a remote procedure specification file. It defines functions that will be remotely executed functions.

Functions are restricted: they may take at most one *in* parameter, and return at most one *out* parameter as the function result.

If you want to use more than one in parameter, you have to wrap them up in a single structure, and similarly with the out values.

Multiple functions may be defined at once. They are numbered from one upwards, and any of these may be remotely executed

The specification defines a program that will run remotely, made up of the functions. The program has a name, a version number and a unique identifying number (chosen by you)

For example, a program may have two local functions to find the date on a machine. The local definitions could be

```
long bin_date(void);  
char *str_date(long);
```

The program with these specified as remote procedures for a remote machine would define the two functions `bin_date` and `str_date` in file `rdate.x`:

```
program RDATE_PROG {
    version RDATE_VERS {
        long BIN_DATE(void) = 1;
        string STR_DATE(long) = 2;
    } = 1;
} = 0x20000000; /* max = 0x3FFFFFFF */
```

Each of these could have one argument

# rpcgen

rpcgen is a program that takes a specification file as command line parameter and generates C source files that can be used as client and server stubs

rpcgen run on rdate.x would generate files

- **rdate.h** - a header file for both client and server sides
- **rdate\_svc.c** - a set of stub functions for use on the server side. This also defines a main function that will allow the server side to run as a server program i.e. it can run and handle requests across the network
- **rdate\_clnt.c** - a set of stub functions for use on the client side that handles the remote call

Functions are generated from the specification as follows:

- The function name is all lower-case, with `VERSION_NUMBER` appended
- On the client side the function generated has two parameters, on the server side it also has two parameters, one is a pointer to the parameter in the specification, the other is a pointer to a lower level information structure (not of interest)
- The client side function has either the one parameter of the spec, or a dummy `void * pointer` (use `NULL`) as first parameter
- On the client side, the second parameter is a ``handle" created by the C function `clnt_create()` (port number identification from the binder)
- On both sides, the function return value is replaced by a pointer to that function return value

# Example date.h

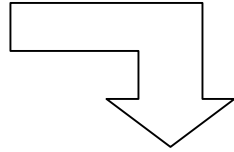
```
/* Please do not edit this file.
 * It was generated using rpcgen.
 */
#ifndef _DATE_H_RPCGEN
#define _DATE_H_RPCGEN
#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

#define DATE_PROG 0x31111111
#define DATE_VERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define BIN_DATE 1
extern long * bin_date_1(void *, CLIENT *);
extern long * bin_date_1_svc(void *, struct svc_req *);
#define STR_DATE 2
extern char ** str_date_1(long *, CLIENT *);
extern char ** str_date_1_svc(long *, struct svc_req *);
extern int date_prog_1_freeresult (SVCXPRT *, xdrproc_t,
caddr_t);
```

# continua



```
#else /* K&R C */
#define BIN_DATE 1
extern long * bin_date_1();
extern long * bin_date_1_svc();
#define STR_DATE 2
extern char ** str_date_1();
extern char ** str_date_1_svc();
extern int date_prog_1_freeresult ();
#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_DATE_H_RPCGEN */
```

# Example date\_clnt.c

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

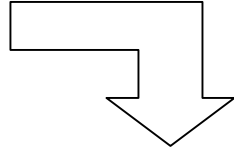
#include <memory.h> /* for memset */
#include "date.h"

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

long *bin_date_1(void *argp, CLIENT *clnt)
{
    static long clnt_res;

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

# continua



```
char **str_date_1(long *argp, CLIENT *clnt)
{
    static char *clnt_res;

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

# Example client.c

```
/*
 * client program for remote date program
 */

#include <stdio.h>
#include <rpc/rpc.h>      /* standard RPC include file */
#include "date.h"        /* this file is generated by rpcgen */

main(int argc, char *argv[])
{
    CLIENT *cli;          /* RPC handle */
    char *server;
    long *lresult;       /* return value from bin_date_1() */
    char **sresult;      /* return value from str_date_1() */

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }

    server = argv[1];

    /*
     * Create client handle
     */
```

```

if ((cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
    /*
     * can't establish connection with server
     */
    clnt_pcreateerror(server);
    exit(2);
}

/* First call the remote procedure "bin_date". */

if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(3);
}
printf("time on host %s = %ld\n",server, *lresult);

/* Now call the remote procedure str_date */

if ( (sresult = str_date_1(lresult, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(4);
}
printf("time on host %s = %s", server, *sresult);

clnt_destroy(cl);          /* done with the handle */
exit(0);
}

```

# Example date\_svc.c

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

#include "date.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>

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

static void date_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        long str_date_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 BIN_DATE:
        _xdr_argument = (xdrproc_t) xdr_void;
        _xdr_result = (xdrproc_t) xdr_long;
        local = (char *(*)(char *, struct svc_req *))
bin_date_1_svc;
        break;

    case STR_DATE:
        _xdr_argument = (xdrproc_t) xdr_long;
        _xdr_result = (xdrproc_t) xdr_wrapstring;
        local = (char *(*)(char *, struct svc_req *))
str_date_1_svc;
        break;

    default:
        svcerr_noproc (transp);
        return;
}
memset ((char *)&argument, 0, sizeof (argument));
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;
}

```

```

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

    pmap_unset (DATE_PROG, DATE_VERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "cannot create udp service.");
        exit(1);
    }
}

```

```
if (!svc_register(transp, DATE_PROG, DATE_VERS, date_prog_1,
IPPROTO_UDP)) {
    fprintf (stderr, "unable to register (DATE_PROG,
DATE_VERS, 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, DATE_PROG, DATE_VERS, date_prog_1,
IPPROTO_TCP)) {
    fprintf (stderr, "unable to register (DATE_PROG,
DATE_VERS, tcp).");
    exit(1);
}

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

# Example server.c

```
/*
 * dateproc.c    remote procedures; called by server stub
 */
#include <rpc/rpc.h>          /* standard RPC include file */
#include "date.h"           /* this file is generated by rpcgen */
/*
 * Return the binary date and time
 */
long *bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
    static long timeval;    /* must be static */

    timeval = time((long *) 0);
    return(&timeval);
}

/*
 * Convert a binary time and return a human readable string
 */

char **str_date_1_svc(long *bintime, struct svc_req *rqstp)
{
    static char *ptr;      /* must be static */
    ptr = ctime(bintime);  /* convert to local time */
    return(&ptr);
}
```