

Fonti:

1. Coulouris G.F., Dollimore J.B. and Kindberg T. (1994). Distributed Systems, Concepts and Design. Addison-Wesley (2nd ed.)
2. Tanenbaum, A.S. (1995). Distributed Operating Systems. Prentice-Hall.

[Pagina intenzionalmente vuota]

Request-reply

Applicazione distribuita: insieme di componenti sw eseguiti su varie macchine.

Architettura client-server di un'applicazione distribuita:

- gli utenti interagiscono con programmi-processi *client*
- i client effettuano computazioni avvalendosi di servizi forniti da processi (anche remoti) *server*
- ogni server è caratterizzato da un insieme di operazioni
- un server può essere client di altri server

Interazione client-server convenzionale (a scambio di messaggi)

- protocollo *request-reply*
- client invia *request* message (con operazione e, possibilmente, dati)
- server effettua calcoli e risponde con *reply* (con codice risultato e, possibilmente, dati)
- client attende reply (anche se l'operazione richiesta è *dataless*)
- in genere, il client imposta per la reply un timeout che può scadere

RPC: introduzione

Le *RPC* (Remote Procedure Call) integrano

- il meccanismo request-reply della programmazione client-server
- in un paradigma linguistico della programmazione convenzionale (non distribuita)

Nozione di RPC:

- il cliente chiede un'operazione invocando una chiamata di procedura, la *RPC*, con eventuali parametri
- ciò causa computazioni, per lo più remote
- il cliente torna dalla chiamata
- il risultato della *RPC* è in: valore restituito, e (eventuali) parametri risultato della *RPC*
- opportuni codici possono indicare fallimento o mancata risposta (entro un timeout)

Implementazione RPC:

- attraverso scambio di messaggi request-reply
- ma scambio (idealmente) trasparente per (programmatori dei) clienti

Trasparenza (a grandi linee):

- linguisticamente la *RPC* è una chiamata di procedura come la *LPC* (Local PC)
- ma il processo che la esegue, rispetto a quello che la chiama, è:
 - *distinto* e
 - (in genere) *remoto*

si vedrà come ciò comprometta in parte la trasparenza.

RPC: introduzione (+)

In pratica, un servizio è caratterizzato da questi aspetti:

1. *gestisce risorse* (p.es. dati) per conto dei clienti
2. rende accessibili le risorse ai clienti solo attraverso un insieme prefissato di *operazioni*
3. è supportato da processi/macchine server, via *request-reply*
4. può *cambiare stato* alle risorse, in risposta a certe operazioni
5. *risorse condivisibili*: ogni cliente ne vede i cambiamenti di stato

La RPC fornisce un modello concettuale per concepire/usare i servizi e strutturare i programmi client:

- un servizio è visto come un modulo SW che esporta un set di procedure adeguate per operare su dati (astratti)
- rispetto alle caratteristiche 1/2, i dati corrispondono alle risorse, le procedure alle operazioni
- esempi:
 - *file service*, visto come insieme di procedure/operazioni sul tipo di dato *file*
 - *authentication service*, visto come insieme di procedure/operazioni sui tipi di dato *username* e *password*

RPC: introduzione (+)

Un servizio/modulo SW convenzionale è

- *statico*: codice oggetto cui le applicazioni si collegano tramite linking statico (in un altro senso) o dinamico

Al contrario, un modulo SW/servizio distribuito è

- *dinamico*: è supportato oltre che da codice, da un processo server che esegue il codice stesso

Per ragioni di efficienza, il processo server ha, in genere, lifetime molto più lunga di ciascun client

Vantaggio: risulta semplice far condividere ai client risorse, cioè dati:

- il dato/risorsa è nello spazio di indirizzamento del processo server
- la risorsa è, inevitabilmente, incapsulata nel codice del server i client, anche volendo, non potrebbero accedere o modificare direttamente la risorsa
- il server esporta RPC per accesso e modifica del dato/risorsa

Al contrario,

- se i client sono processi distinti (come di è supposto)
- e si vuole che condividano risorse
- queste non possono essere (viste come) variabili globali, a causa della separazione di loro address space
- (nella programmazione non distribuita, si ricorre in genere a file condivisi)

La semantica va definita/giudicata rispetto all'affinità a quella delle LPC.

L'obiettivo è la *trasparenza* = affinità completa.

Aspetti principali della semantica:

1. parametri *in*, *out* e misti
implementazione: valore del parametro copiato in request e/o reply
2. semantica dei parametri *in* come per LPC
3. parametri puntatore/*var*: l'indirizzo non ha senso su un'altra macchina; si copia l'oggetto puntato su messaggio request o reply;
per decidere request/reply occorre sapere se il parametro è *in* o *out*;
⇒ spesso, il linguaggio ospite non ha la nozione di parametro in/out;
serve quindi anche un *IDL*: Linguaggio di Definizione Interfacce / rappresentazione di dati,
4. variabili globali (comuni a client/server) inaccessibili da codice RPC
5. parametri e risultato non possono contenere puntatori alla memoria (p.es. *struct* con elementi puntatore)

La limitazione 5 (pag. 94) per cui parametri e risultato non possono contenere puntatori non è in pratica grave.

Caso tipico: parametri e/o risultato sono una struttura dati dinamica (lista/grafico/albero).

Soluzione:

- la struttura dati andrebbe comunque vista come ADT dai clienti;
⇒ i puntatori non dovrebbero figurare come (concettualmente) tali tra i parametri *in* nell'invocazione di operazioni
- la RPC può restituire come risultato una *opaque reference*, legale solo come argomento di successive RPC
in pratica si tratta, dato un oggetto remoto *x* (lato server),
 - in Java/RMI, di una reference *x* che il cliente non può usare per accedere direttamente ai membri dell'oggetto remoto *x*
 - in C/RPC, di un puntatore a *x*, sia *p*, a cui il cliente evita di applicare dereferencing (**p*)
- quindi in genere la struttura dati "risiede" dal lato server e figura solo come reference tra i parametri delle RPC del client

Talora, però, la struttura dati deve "viaggiare", cioè essere copiata in un messaggio request o reply.

Esempio: lista concatenata *list* che risiede:

- dal lato client, che la usa come parametro *in* in una RPC;
- oppure dal lato server, che ne restituisce, come risultato di una RPC, il valore (e non una reference)

In tal caso la struttura dati, prima di essere copiata nel messaggio, subisce un *flattening* o *serialization* (es. alberi codificati con parentesi).

Spesso l'architettura del SW cliente ha un livello extra (*user package*):

Client program	<i>User package interface</i> ↓
User package	<i>RPC interface</i> ↓
Client stub procedures	
Communication handler	

Motivazioni:

- l'interfaccia RPC può dipendere da architettura, SO, . . .
ciò rende i programmi client dipendenti
- accesso ai servizi al livello di astrazione più alto possibile
- implementare funzionalità assenti nei layer RPC, ma che si vogliono offrire ai client program;
p.es. individuazione in rete dei server per un dato servizio
- schermare i client dalla definizione delle interfacce nel linguaggio specializzato RPC

Analogia: in un SO dualismo tra

- (interfaccia delle) system call (p.es. `write`)
- API (p.es. `file/stream` e `printf`)

Ottimizzazioni:

- implementare/duplicare funzionalità proprie del servizio nell'*user package*
- scopi: migliorare performance e/o semplificare il servizio
- limitazioni: a task che non compromettono affidabilità e sicurezza per altri clienti
(tipicamente *user package* è meno *trusted* di layer RPC)

Classi di sistemi RPC

1. Sistema RPC integrato con un linguaggio di programmazione; di questo fa parte integrante il linguaggio IDL. Esempi:
 - Cedar: linguaggio Mesa
 - Arjuna
 - Argus: linguaggio CLU, con costrutti detti *guardian* (ADT) e *handler* (operazioni ADT)
2. Sistema RPC con linguaggio IDL ad hoc per descrivere l'interfaccia (operazioni) del server rispetto al client.

Vantaggio: flessibilità rispetto all'ambiente di sviluppo (ma quasi sempre è il C!)

Esempi:

- RPC Sun
- ANSA
- Matchmaker (Mach)
- MIG (Mach Interface Generator)

IDL: Interface Definition Language

Ogni servizio RPC *esporta* (rende disponibili) ai clienti certe operazioni (altrettante RPC).

Un IDL serve a definire:

- *nome* di un servizio RPC
- (spesso) *versione* del servizio
- per ogni operazione del servizio, *segnatura* o *prototipo*, cioè:
 - *nome*
 - *tipi e direzione (in/out)* dei parametri

Esempio (RPC Sun):

```
/* arit.x */  
  
program ARIT_PROG { // remote module name  
    version ARIT_PROG_V {  
        int inc(int) = 1; // 1 is rpc num id  
        int dbl(int) = 2; // 2 is rpc num id  
    } = 33; // version number  
} = 0x20000001; // program number */
```

Un servizio RPC descritto in IDL costituisce l'input per un *interface compiler* (es. `rpcgen` di Sun RPC).

L'interface compiler legge il file di definizione dell'interfaccia e:

- genera automaticamente due moduli detti

client stub:

- quando il cliente invoca la RPC $f()$
- esso invoca, *in concreto*, una $f()$ dello stub (*in astratto*, ovviamente, la $f()$ invocata è remota)

server stub o skeleton:

- quando un cliente invoca la RPC $f()$,
- in concreto dal lato server sarà lo stub, attivato dal messaggio *request* ricevuto, a invocare la $f()$ corrispondente

quindi gli stub, generati automaticamente, nascondono al programmatore di applicazioni il lavoro sporco (v. p. 102)

- genera le dichiarazioni necessarie di tipi e procedure nel linguaggio target, traducendo dall'IDL

(tipicamente in C si tratta di header file)

Un IDL compiler per linguaggi target multipli consente client e server scritti in linguaggi diversi.

In definitiva: l'IDL è usato come un esperanto dei linguaggi di programmazione, per la parte di definizione di prototipi e tipi.

```
/* arit.x */

program ARIT_PROG {           // remote module name
    version ARIT_PROG_V {
        int inc(int) = 1;    // 1 is rpc num id
        int dbl(int) = 2;    // 2 is rpc num id
    } = 33;                  // version number
} = 0x20000001;              /* program number */

/* arit.h */
/* generated from arit.x using rpcgen, edited by hand */

#ifndef _ARIT_H_RPCGEN
#define _ARIT_H_RPCGEN

#include <rpc/rpc.h>

#define ARIT_PROG 0x20000001
#define ARIT_PROG_V 33

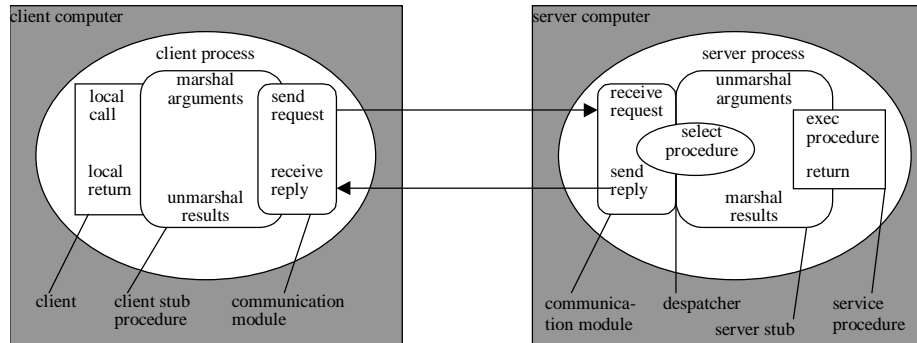
#define inc 1 // rpc id
extern int * inc_33(int *, CLIENT *);
extern int * inc_33_svc(int *, struct svc_req *);

#define dbl 2 // rpc id
extern int * dbl_33(int *, CLIENT *);
extern int * dbl_33_svc(int *, struct svc_req *);

extern int arit_prog_33_freeresult
    (SVCXPRT *, xdrproc_t, caddr_t);

#endif /* !_ARIT_H_RPCGEN
```

Implementazione RPC: lato client

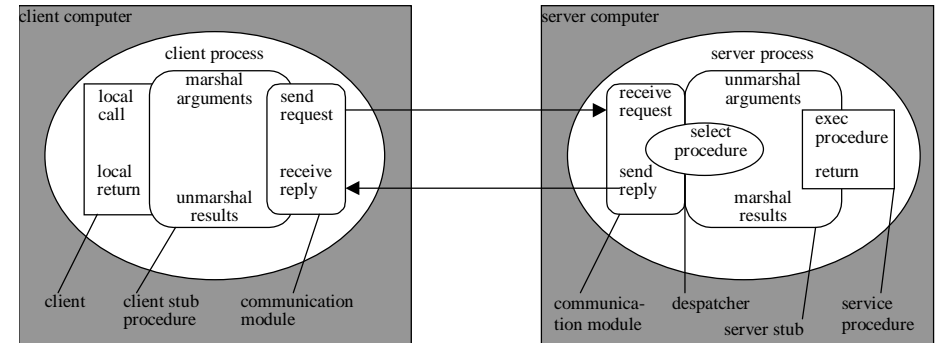


Dal lato client:

- il *client* è il codice materialmente scritto dal programmatore di applicazioni distribuite;
la RPC $f()$ è remota in astratto, ma in concreto non può che essere implementata con una LPC $f()$
- il *client stub*:
 - contiene gli entry point delle LPC corrispondenti alle RPC del client
 - effettua il *marshalling*: serializza in una stringa di bit i parametri della RPC
 - prepara la *request*, inserendovi i “marshalled parameters”
- il *communication module* invia la *request* e attende la *response*, che ripassa allo stub come stringa di bit
- lo *stub* estrae (“unmarshals”) dalla *response* il risultato delle RPC, convertendolo da stringa di bit in dato di tipo appropriato
- adesso, nel client, ritorna $f()$, nella duplice accezione di RPC/LPC

NB: le interazioni tra moduli sono chiamate di funzione locale.

Implementazione RPC: lato server



Dal lato server, un processo server è in esecuzione ed è in grado di servire richieste multiple, da più clienti:

- il *communication module* aspetta
- il *server stub*
- la *service procedure*

NB:

- le interazioni tra moduli sono chiamate di funzione locale
- server parallelo?

RPC: semantica e garanzie di esecuzione

Gli ambienti distribuiti sono *error prone*.

Tecniche di controllo dei problemi di comunicazione (*delivery guarantees*):

request retry - il protocollo RPC riprova a spedire la *request* finché ottiene una risposta

duplicate request filtering - il lato server riconosce *request* già ricevute

reply caching and reuse - logica conseguenza del precedente

Adottare o no queste tecniche, dà luogo a varie *semantiche* per la RPC:

Delivery guarantees			RPC call semantics
Retry request message	Duplicate request filtering	Redo computation or Resend reply	
No	Not applicable	Not applicable	<i>Maybe</i>
Yes	No	Redo computation	<i>At least once</i>
Yes	Yes	Resend reply	<i>At most once</i>

Accettabilità delle semantiche:

maybe - quasi sempre inaccettabile

at least once - accettabile solo per operazioni *idempotenti* (es. *read*)

at most once - la più *trasparente*. Più precisamente:

- RPC ritorna normalmente: *exactly once*
- RPC ritorna anormalmente (timeout): *once or nonce*

Exception Handling

Una RPC può fallire per varie ragioni:

- superamento n. max timeout/retry, a causa di:
 - server failure
 - server troppo carico
 - perdita di request o reply
- errore del codice rpc lato server (p.es. file name non valido)

Tre approcci al trattamento delle failure.

exception handling - usato in ambienti RPC il cui linguaggio preveda:

- definizione di exception handlers, per varie classi di eccezioni (legate ognuna a specifiche modalità di failure)
- *exception raising*, cioè invocazione automatica, quando si verifica un'eccezione' dell'handler relativo

Esempi: Argus (linguaggio CLU), Cedar/Mesa, RMI/Java.

In ambienti RPC language independent, l'IDL deve permettere la definizione di eccezioni/handler.

errori come tipi di dato - L'interfaccia di un modulo RPC include:

- oltre a usuale definizione di nome/tipo/argomenti operazione
 - definizione del tipo degli eventuali errori restituiti; p.es. il tipo degli errori può essere enumerato o una `struct`
- Esempio: Courier RPC (successore di Cedar)

nessun meccanismo specifico (p.es. RPC Sun) - RPC che fallisce:

- restituisce valore di errore (-1 / NULL) anziché valore atteso
- lascia codice di errore specifico in una var globale

Svantaggio: codice disseminato di test su errori.

Trasparenza

- trasparenza = grado di approssimazione rispetto a LPC.
- è l'obiettivo del progetto di un sistema RPC, ma per varie ragioni è soggetta a compromessi

Misure per la trasparenza (proposte per la 1a volta in Cedar):

- request retry trasparente
- sintassi trasparente (grazie al marshalling)

Argomenti contro la trasparenza (Argus):

- computazioni RPC durano molto di più (vari ordini di grandezza)
- RPC più *error prone*

Quindi, nella filosofia di Argus:

- è bene che il programmatore sia consapevole che usa RPC
- questo deve riflettersi in una sintassi RPC non trasparente (cioè che estende effettivamente il linguaggio)
- il cliente deve poter abortire una RPC (p.es. che dura troppo)
in questo caso, lo stato del server deve tornare quello di partenza;
quindi occorrono meccanismi di supporto del "roll-back"