Università degli Studi di Bologna
Scuola di Ingegneria

Corso di
Reti di Calcolatori T

Sistemi di Chiamate Remote

Antonio Corradi

Anno Accademico 2023/2024

SEMANTICA DELLA COMUNICAZIONE

Internet propone dei **protocolli standard** **se rappresentano un accordo degli utenti** ossia che possano essere sempre accettati dalla comunità di uso e condiziona la loro standardizzazione solo se sono **provati dalla comunità di utenti**

Si definiscono le **Request for Comments** (o **RFC**) che sono documenti di **proposta di standard** che diventano standard se approvati dalla comunità di utilizzo

https://www.rfc-editor.org/

Ci sono gli standard accettati, i draft, le prassi …

In generale la comunità Internet ha sempre mirato a implementazioni di **protocolli aperti, efficienti e capaci di fornire soluzioni che siano anche molto usabili e semplici**

SEMANTICA DELLA COMUNICAZIONE

Internet risponde alla filosofia di massima usabilità su ogni macchina e con scarse risorse

- **MAY-BE (o BEST-EFFORT)**

Per limitare i costi ci si basa su un **solo invio asincrono di ogni datagramma / informazione** ⇨ il messaggio **può arrivare o meno**

Quindi non si fanno azioni di recovery per mantenere il costo basso

IL PROGETTO INTERNET è tutto BEST-EFFORT

- rappresentato da IP in cui ogni azione è fatta una volta sola, senza preoccuparsi di qualità, di affidabilità e di garanzie
- UDP rappresenta il protocollo di Trasporto end-to-end in cui per ottenere bassi costi non si fanno azioni per garantire affidabilità

Lo standard sacrifica la **qualità del servizio** (**Quality of Service o QoS**) alla applicabilità globale in una visione poco aziendale **ma molto aperta (open science – codice visibile e provabile)**

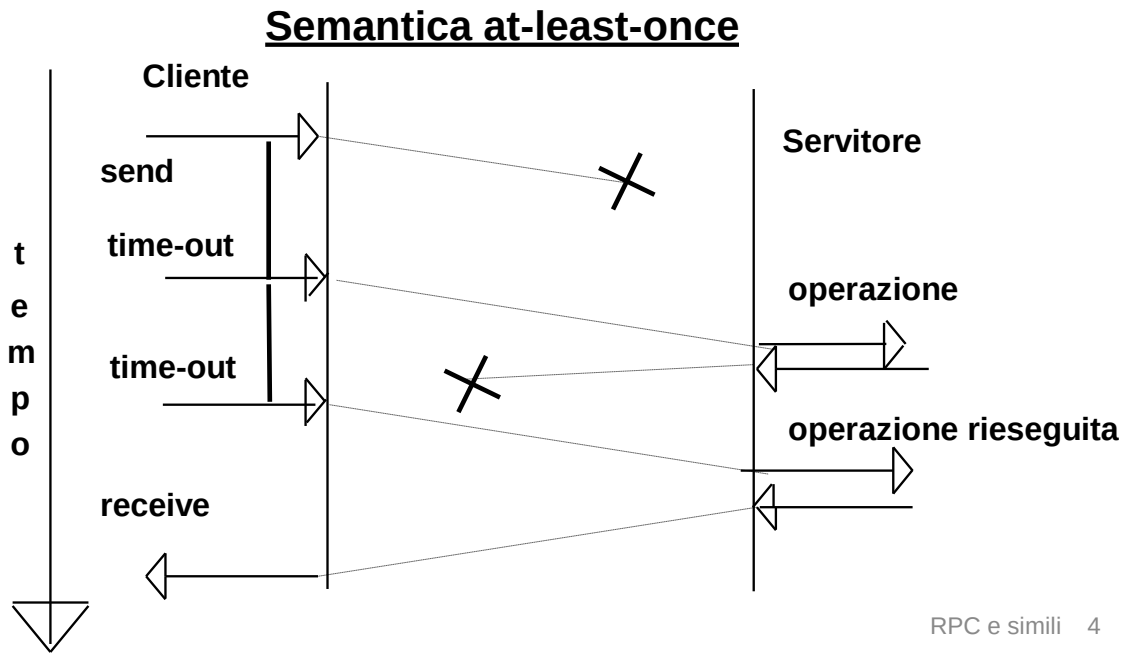
SEMANTICA AT-LEAST-ONCE

Internet deve anche fornire **altre risposte e maggiore qualità**, e tenere conto di informazioni ricevute e ritrasmissioni successive a timeout

SEMANTICA AT-LEAST-ONCE
prevedendo ritrasmissioni ad intervallo da parte del mittente

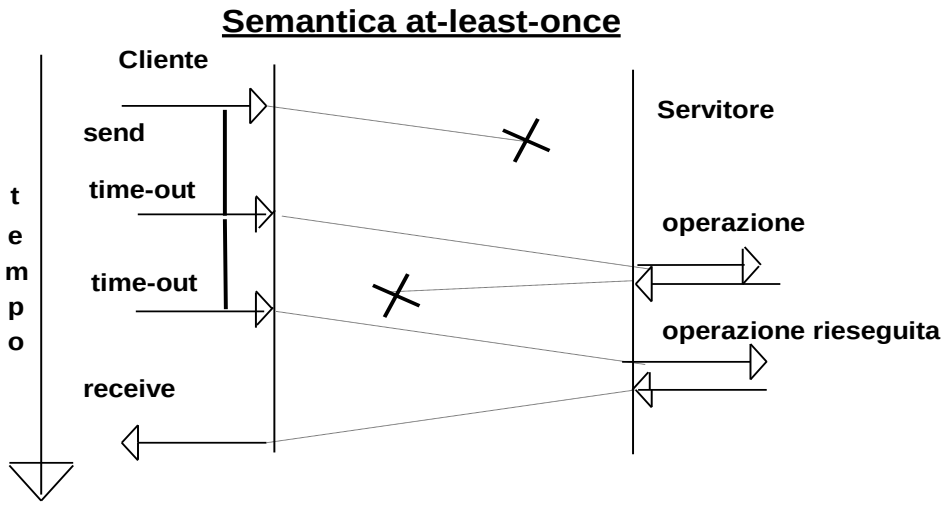
- **il messaggio può arrivare anche più volte a causa dei messaggi duplicati dovuti a ritrasmissioni da parte del mittente**

Cominciamo a ragionare in termini di Cliente / Servitore



AT-LEAST-ONCE: AZIONI MITTENTE

Semantica adatta per azioni idempotenti
in caso di insuccesso nessuna informazione



implementazione

PROGETTO RELIABLE (AL MITTENTE)

- il cliente fa ritrasmissioni (quante?, ogni quanto? ...)
- il server non se ne accorge
- **IL CLIENTE SI PREOCCUPA DELLA AFFIDABILITÀ**
- Il cliente decide (in modo unilaterale) la durata massima

IL SERVER NON SE NE ACCORGE E RIESEGUE

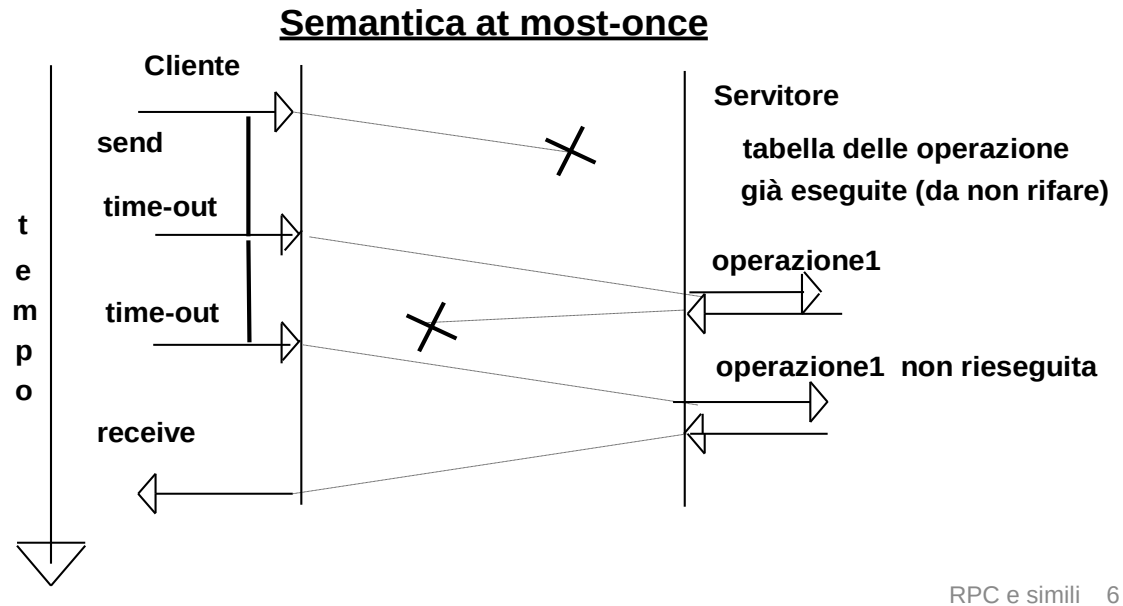
SEMANTICA AT-MOST-ONCE

AT-MOST-ONCE (Più invii ad intervalli e anche Stato sul Server) – intervento su Cliente e Servitore

CLIENTE e SERVITORE lavorano in modo coordinato per ottenere garanzie di correttezza e affidabilità

- il messaggio, se arriva, viene considerato al più una volta dal servitore
- la semantica non introduce vincoli sulle azioni applicative (idempotenza)

Anche qui in caso di insuccesso nessuna informazione

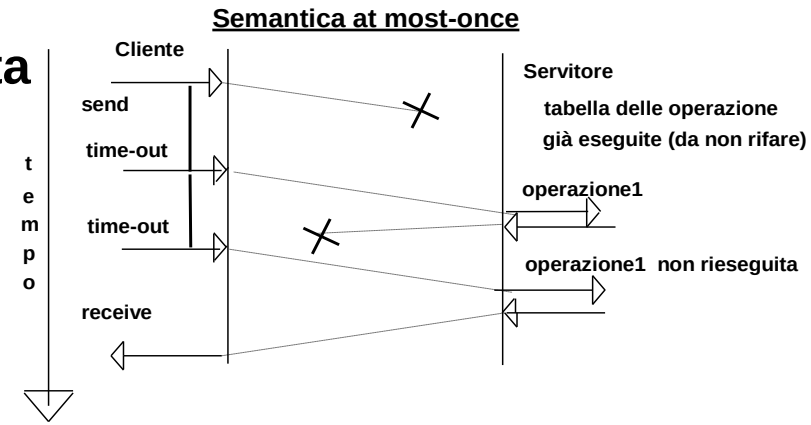


SEMANTICA AT-MOST-ONCE: TCP

Implementazione **PROGETTO RELIABLE per cliente e servitore**

- **il cliente fa ritrasmissioni** (decise dal protocollo in TCP)
- **il server mantiene uno stato per riconoscere i messaggi già ricevuti e per non eseguire azioni più di una volta**

Dimensionamenti in tempi e risorse



STATO MANTENUTO PER UN CERTO TEMPO

- **Si noti la durata della azione per le due parti**
- Il cliente decide la durata massima della propria azione
- Il server mantiene uno stato per garantire correttezza
- Per quanto tempo i pari mantengono lo stato della interazione? E se uno fallisce? E se non si coordinano?

SEMANTICA EXACTLY-ONCE: DI PIÙ

Semantica at-most e at-least once

- Se le cose vanno male manca il coordinamento e non sappiamo **cosa sia successo a entrambi** (decisioni unilaterali di ciascuno)

A livello applicazione spesso si vuole una maggiore garanzia, EXACTLY-ONCE O ATOMICITÀ

- **il messaggio arriva una volta sola per tutti e due oppure**
- **il messaggio o è arrivato o non è stato considerato da entrambi**

⇨ semantica molto **coordinata sullo stato**

Al termine **i pari sanno se l'operazione è stata fatta o meno**

- i pari lavorano entrambi per ottenere il massimo dell'accordo e della reliability

PROGETTO con completa conoscenza dello stato finale

- **AFFIDABILITÀ e COORDINAMENTO massimo**

Semantica TUTTO o NIENTE

ATOMICITÀ O TUTTO O NIENTE

La semantica TUTTO O NIENTE non ha una durata massima prevedibile…

In caso le cose vadano bene

- il messaggio arriva una volta e una volta sola viene trattato, riconoscendo i duplicati (tutto ok)

In caso le cose vadano male

- il cliente e il servitore sanno se il messaggio è arrivato (e considerato 1 volta sola - **tutto**) o se non è arrivato
- Se il messaggio non è arrivato a uno dei due, il tutto deve essere riportato indietro (**niente e undo**)

Completo coordinamento delle azioni ai partner, ma durata delle azioni non predicibile

Durata massima, anche non limitatanel caso peggiore

- Se uno dei due fallisce, bisogna aspettare che abbia fatto il recovery. Solo allora entrambi sanno realmente come è andata (tutto o niente)

SEMANTICA PROTOCOLLI TCP/IP

SEMANTICA operazioni dei protocolli (dovuta a IP)

UDP e IP may-be

l'azione può essere stata fatta o meno

TCP at-most-once

l'azione può essere avvenuta al più una volta

IN CASO DI INSUCCESSO

- **NESSUNA GARANZIA DI ACCORDO sullo STATO della comunicazione e dei partecipanti**

La semantica decisa consente di mantenere accettabile **sia la durata delle operazioni e sia il carico e costo dei protocolli** (in termini di banda richiesta e risorse dedicate)

Ancora di più, per operazioni di gruppo (multicast e broadcast)

SEMANTICA TRASPORTO

TCP/UDP - protocolli di livello Trasporto (TX)
Servizio con connessione (TCP) e senza (UDP)

TCP **protocollo connesso**

- **Connessione bidirezionale tra gli endpoint**
- **Dati differenziati (normali senza limite di banda e prioritari a banda limitata)**
- **Controllo di flusso a byte**
  - Ordine corretto dei byte, ritrasmissione messaggi persi
- **Controllo di flusso tra i due partecipanti**
  - bufferizzazione
- **Semantica at-most-once in caso di successo**

  con l'obiettivo di consentire **costi bassi, durata limitata in TX,** e di avere possibilità di gestire eccezioni in modo più trasparente possibile

  in caso di insuccesso non exactly-once: non prevedibile

REMOTE PROCEDURE CALL - RPC

REMOTE PROCEDURE CALL (RPC) come **estensione del normale meccanismo di chiamata a procedura locale**, come **cliente/servitore nel distribuito**

Approccio **applicativo** di alto livello (**livello 7** di OSI)

- **il cliente invia la richiesta ed attende in modo sincrono fino alla risposta fornita dal servitore stesso**

Differenze rispetto alla chiamata a procedura locale

- sono coinvolti **processi distinti** su nodi diversi
- i processi **cliente e servitore** hanno vita separata
- i processi **non** condividono lo **spazio di indirizzamento**
- sono possibili **malfunzionamenti** sui **nodi** o nella **infrastruttura di comunicazione**

CHIAMATE REMOTE

Le realizzazioni della chiamate remote rendono un servitore ed i clienti adatti per la realizzazione nel distribuito e quindi non possono più essere usate nel normale meccanismo di chiamata a procedura locale

Usando un approccio **applicativo** tutto l'ambiente di supporto cambia rispetto al locale e focalizza tutta la parte di superamento del vincolo della località nel modo più efficiente

- i processi **non** condividono lo **spazio di indirizzamento e usano nuove modalità**

Le funzioni remotizzate non possono essere chiamate in modo locale (ma sempre come se fossero remote)

RPC: PROPRIETÀ

Remote Procedure Call

- consente il **controllo dinamico del tipo** dei parametri e del risultato
- include il **trattamento dei parametri** di ingresso / uscita dal cliente al servitore (e viceversa) detto marshalling o almeno serializzazione

(**livello di presentazione: marshalling**)

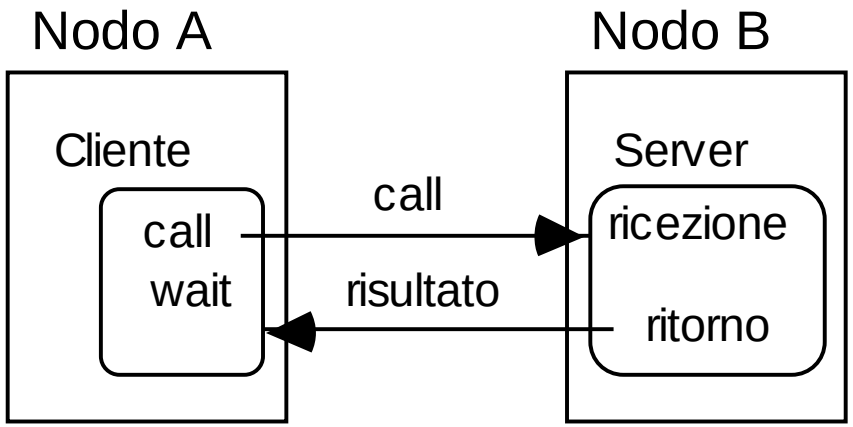
PROPRIETÀ - uniformità totale impossibile (visibilità guasti)

- **trasparenza** approccio locale e remoto
- **type checking** e **parametrizzazione**
  lo stesso controllo di tipo e dei parametri
- **controllo concorrenza** e **eccezioni**
- **binding distribuito**
- possibile **trattamento degli orfani**
  orfano il processo server che non riesce a fornire risultato
  e anche il cliente senza risposta

RPC – STORIA

Prima visione sistematica dovuta a **Birrel** e **Nelson** (1984) partendo da quanto usato in Xerox, Spice, Sun, HP, etc.

- Molte implementazioni
- Default: cliente sincrono bloccante



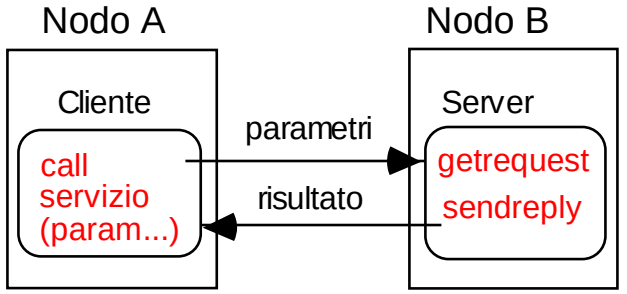
Progetto Athena MIT, ITC CMU, ...
RPC usando scambi di messaggi coordinati

CLIENTE	SERVITORE
send	**get-request** <operazione>
wait	**send-reply**

Si devono prevedere anche molte azioni di supporto:
trasformazioni di dati, nomi, controllo esecuzione e dati, ...

RPC – PRIMITIVE

Ogni sistema usa primitive diverse senza troppe regole standard



API

Dalla parte del **cliente**

- **call servizio** (servitore, argomenti, risultato)

Dalla parte del **servitore**

- **getrequest – sendreply**

Due tipi di politiche di **accettazione** e di **risposta** con **diverse** possibilità di **concorrenza sul server**:

- il servizio svolto da un **unico processo che decide quando e se eseguire i metodi** (modo **esplicito** e **sequenziale**)
- ogni servizio è eseguito da **processo indipendente**, generato per ogni richiesta automaticamente (approccio **implicito** e **parallelo**)

RPC – TOLLERANZA AI GUASTI

Obiettivo applicativo è di **mascherare i malfunzionamenti**

- perdita di messaggio di richiesta o di risposta
- **crash del nodo del cliente**
- **crash del nodo del servitore**

Ad esempio, in caso di crash del servitore prima di avere fornito la risposta o in caso di sua non presenza, o in caso di perdita di messaggio in andata o ritorno

Il cliente può tentare politiche diverse e diversi comportamenti:

- **aspettare per sempre**
- **time-out e ritrasmettere** (uso identificatori unici)
- **time-out e riportare una eccezione al cliente**

Spesso **si assume che le operazioni siano idempotenti** ossia

- che si possano eseguire un numero qualunque di volte con lo stesso esito per il sistema cliente/servitore (?)

FAULT TOLERANCE E SEMANTICA

Le RPC hanno alcune semantica tipiche e strategie relative

- **may-be** time-out per il cliente
- **at-least-once** time-out e ritrasmissioni (idempotenza)
- **at-most-once** tabelle delle azioni effettuate
- **exactly-once** l'azione viene fatta fino alla fine

Invece, in caso di crash del cliente, si devono trattare **orfani sul nodo servitore**, ossia i processi in attesa di consegnare risultato

Politiche tipiche

- **sterminio**: ogni orfano risultato di un crash viene distrutto
- **terminazione a tempo**: ogni calcolo ha una scadenza, oltre la quale è automaticamente abortito
- **reincarnazione (ad epoche)**: tempo diviso in epoche; tutto ciò che è relativo alla epoca precedente è obsoleto e distrutto

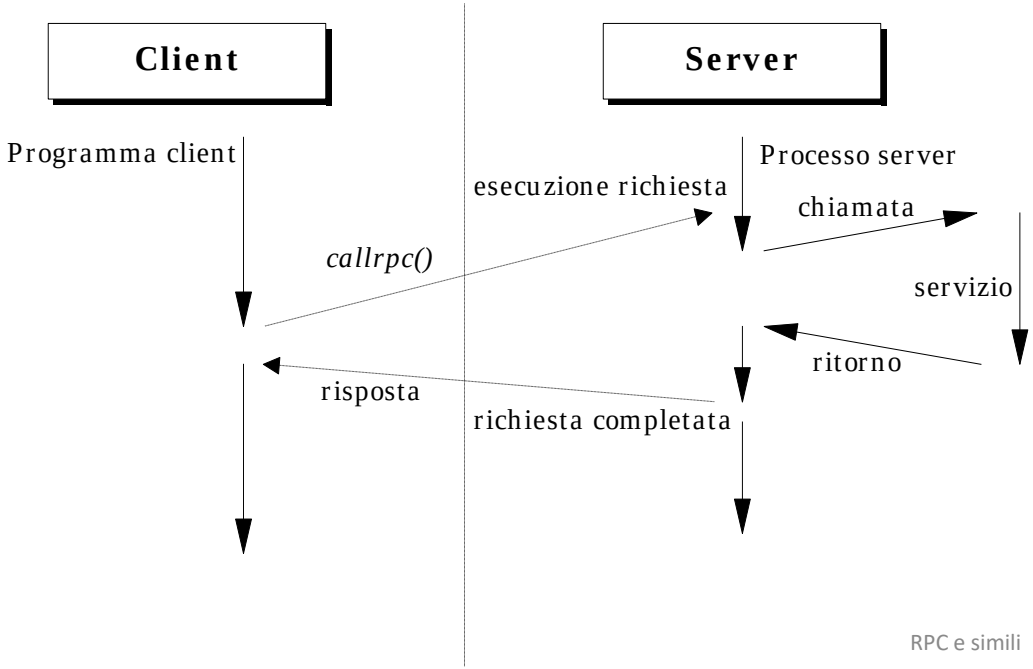
OPEN NETWORK COMPUTING ONC

Sun propone una **chiamata RPC** (disomogenea dalla locale)

primitiva callrpc (…) con parametri aggiuntivi oltre a quelli logici

- nome del nodo remoto
- identificatore della procedura da eseguire
- specifiche di trasformazione degli argomenti (si introduce un formato standard eXternal Data Representation XDR)

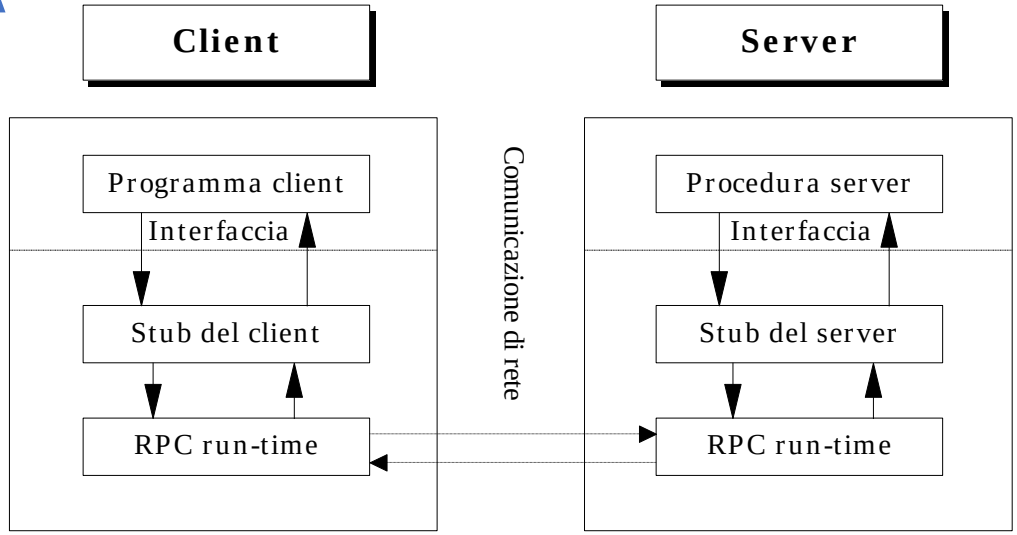
Schema del modello ONC
NON TRASPARENZA
adottato da
HP-UX, **Sun**
Sistemi UNIX compatibili,
Novell Netware



NETWORK COMPUTING ARCHITECTURE NCA

Si introducono ai due endpoint di comunicazione delle **routine stub** per ottenere la **trasparenza**

Le chiamate diventano del tutto locali all' endpoint e stub e si garantisce **TRASPARENZA**



Schema del modello NCA

Gli **stub sono forniti dall'implementazione e generati automaticamente**

Le parti logiche di programma sono "del tutto" inalterate

ci si dirige verso lo stub

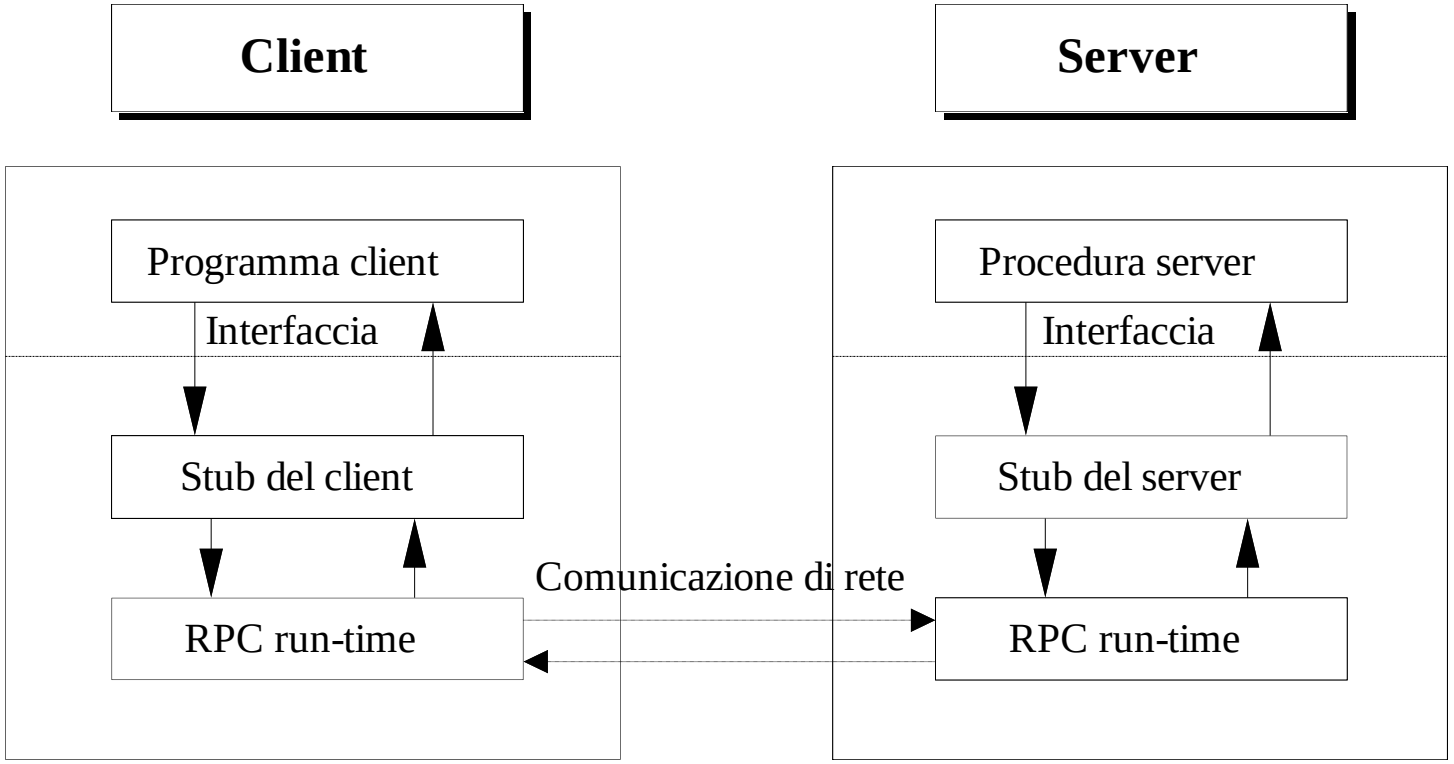
che nasconde le operazioni ⇨ **COSTO: due chiamate per RPC**

Mercury ottimizza i messaggi con stream di chiamate

MODELLO CON TRASPARENZA

Nelson descrive il **modello implementativo NCA trasparente** con **uso di stub** ossia interfacce locali per la trasparenza

- che trasformano la richiesta da locale a remota



Modello **asimmetrico a molti clienti/ un solo servitore**

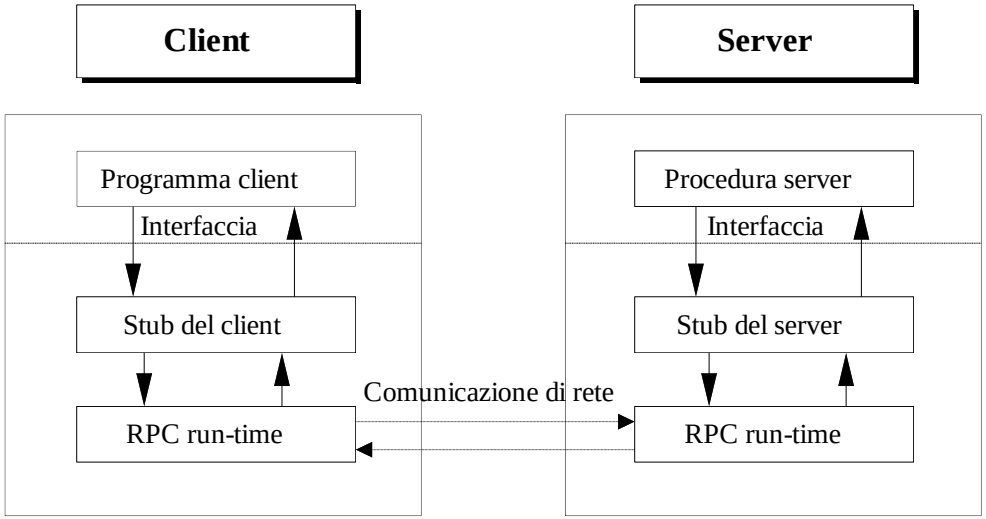
MODELLO CON STUB

Modello **cliente / servitore asimmetrico applicativo**

- Il cliente invoca uno **stub**, che si incarica di tutto: dal recupero del server, al trattamento dei parametri e dalla richiesta al supporto run-time, al trasporto della richiesta
- Il servitore riceve la richiesta dallo **stub** relativo, che si incarica del trattamento dei parametri dopo avere ricevuto la richiesta pervenuta dal trasporto. Al completamento del servizio, lo stub rimanda il risultato al cliente

Lo sviluppo prevede la massima trasparenza

1. STUB prodotti in **modo automatico**
2. L'utente finale progetta e si occupa solo delle **reali parti applicative e logiche**



ESEMPIO DI FUNZIONI DELLO STUB

Negli stub si concentra tutto il **supporto nascosto all'utente finale**

operazione(parametri)

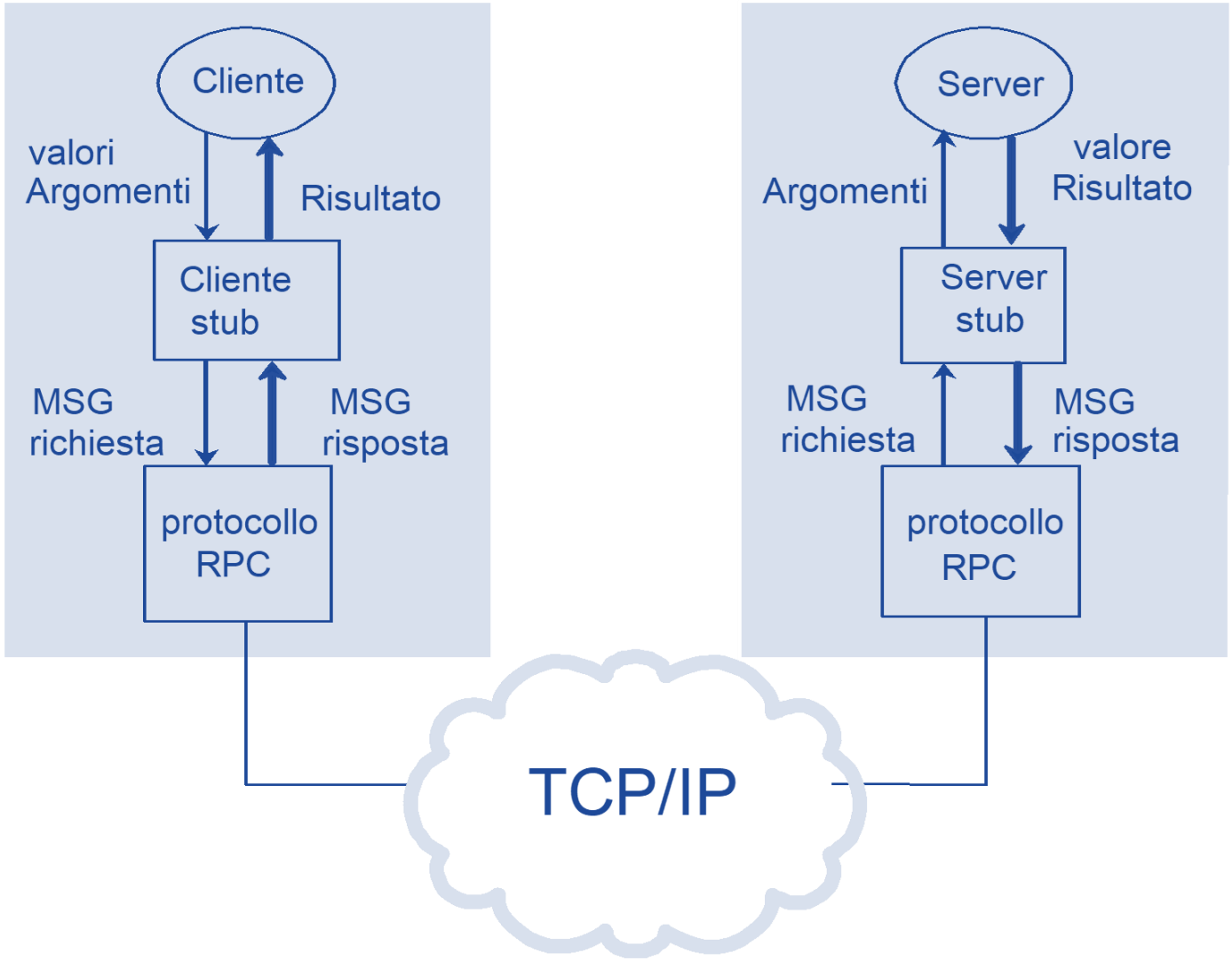
stub cliente:	**stub servitore:**
< ricerca del **servitore**>	< attesa della richiesta>
<***marshalling*** argomenti>	<***unmarshalling*** argomenti>
<send richiesta>	invoca operazione locale
<receive risposta>	ottieni risultato
<***unmarshalling*** risultato>	<***marshalling*** del risultato>
restituisci risultato	<send risultato>
fine stub cliente;	**fine stub servitore;**

operazione(parametri)

In questo schema mancano le **ritrasmissioni possibili** da parte del cliente e il **controllo della operazione** da parte del servitore (concorrenza o meno)

ANCORA SCHEMA DI STUB

Gli stub si occupano ‘automaticamente’ di tutta la **parte distribuita**



PASSAGGIO DI PARAMETRI

I parametri devono passare tra ambienti diversi

Passaggio **per valore** o **per riferimento**

- **In genere visto che non abbiamo memoria condivisa, si ragiona con un passaggio per valore (dal chiamante al chiamato)**
- **Passaggio per riferimento richiede un nuovo approccio (RMI)**

Trattamento default dei **parametri per valore**

- **impaccamento** dei parametri (**marshalling**) e **disimpaccamento** (**unmarshalling**) con dipendenza dal linguaggio utilizzato

Per tipi **primitivi** o una entità con **valori privati**

- **marshalling / unmarshalling** per la presentazione

Per tipi **utente costruiti e dinamici**, ad esempio, una lista o un albero (e memoria dinamica), il contenuto guida la trasformazione

- si deve (?) **copiare** o (?) **muovere** il contenuto e ricostituirlo sul server (per poi riportarlo sul nodo iniziale? O meno)

PASSAGGIO DI PARAMETRI

Passaggio parametri dal cliente al servitore

nel caso di **passaggio per valore**

- **passaggio con trasferimento e visita** (**valore perso sul server**)

nel caso di **passaggio per riferimento**

- **passaggio senza trasferimento** ma rendendo **l'oggetto remoto**
- uso di oggetti che rimangono nel **nodo di partenza e devono essere identificati in modo unico nell'intero sistema**
- Se si vuole riferire un'entità del cliente, si passa **il riferimento alla stessa entità che i nodi remoti possono riferire attraverso RPC**

Per esempio, un oggetto che sia già in uso sul nodo del cliente deve potere **essere riferito dal nodo servitore** e cambiato in **stato senza interferire con l'uso locale dell'oggetto**

TRATTAMENTO DELLE ECCEZIONI

Le RPC hanno previsto integrazione con exception handling

- trattando gli eventi anomali dipendenti dalla distribuzione o ai guasti integrandola con la gestione locale e inserendola nella gestione locale delle eccezioni
- In genere, si specifica la azione per il trattamento anomalo in un opportuno gestore della eccezione

Si può anche inserire l'eccezione nello scope di linguaggio

- CLU (Liskov) a livello di invocazione della RPC
- MESA (Cedar) a livello di messaggio
- Java definizione delle exception

una RPC può produrre il servizio con successo o produrre insuccesso e determinare una eccezione locale con semantica tipica

may-be	con time-out per il cliente
at-least-once	**SUN RPC**
at-most-once	con l'aiuto del trasporto

INTERFACE DEFINITION LANGUAGE

Interface Definition Language IDL

linguaggi per la descrizione delle operazioni remote, la specifica del servizio (detta firma) e la generazione degli stub

Un IDL deve consentire la identificazione non ambigua

- **identificazione (unica) del servizio tra quelli possibili**
  - uso di nome astratto del servizio spesso prevedendo versioni diverse del servizio
- **definizione astratta dei dati da trasmettere in input ed output**
  - uso di un linguaggio astratto di definizione dei dati (uso di interfacce, con operazioni e parametri)

Possibili estensioni: linguaggio dichiarativo con ereditarietà, ambienti derivati con binder ed altre entità

Dalla definizione del linguaggio IDL fornita dall'utente si possono generare gli stub

TIPICI LINGUAGGI IDL

Sono stati definiti molti linguaggi IDL
che hanno permesso di studiare le implicazioni e le politiche a livello di sistema di supporto

Sono nati molti IDL e relativi strumenti correlati per lo sviluppo automatico di parte dei programmi direttamente dalla specifica astratta, esempi sono

SUN	**XDR**	primo esempio di standard proprietario
OSF	**DCE IDL**	
ANSA	**ANSAware**	
HP	**NCS IDL**	
CORBA	**IDL**	

Le differenze hanno generato dibattiti e confronti anche accesi, spentisi visti gli scope limitati dei linguaggi stessi ⇨

Mancanza di standard ☹

ESEMPIO DI XDR

XDR eXternal Data Representation

- Il formato XDR definisce le **operazioni remote** e tutto quello che è necessario conoscere per la generazione di stub (**parametri**)
- L'utente deve sviluppare un file di descrizione logica dei servizi offerti da cui si possono generare gli stub
- Si prevedono più servizi in **versioni diverse** e **tipi primitivi e anche definiti dall'utente**

file msg.x

```
program MESSAGEPROG {
        version MESSAGEVERS {
                int PRINTMESSAGE(string) = 1;
                } = 1;
        } = 0x20000013;
```

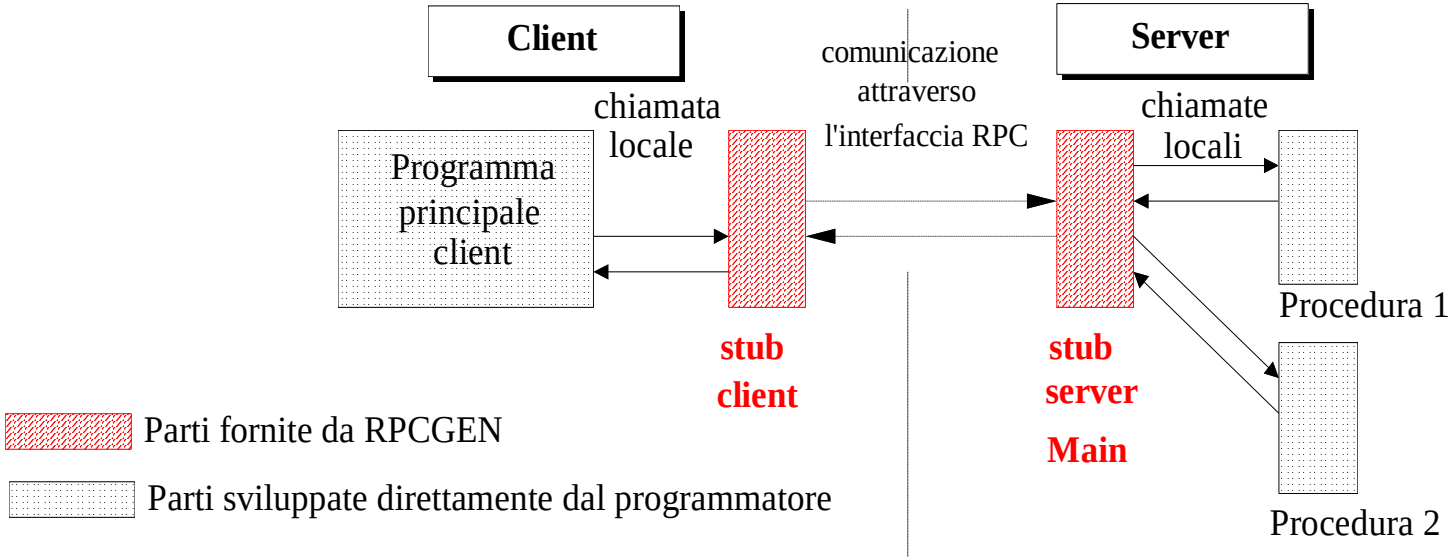
DA IDL A STUB

Gli IDL hanno lo scopo di supporto allo sviluppo dell'applicazione permettendo di generare automaticamente gli stub dalla interfaccia specificata dall'utente

Strumento RPCGEN (Remote Procedure Call Generator)

compilatore di protocollo RPC per generare procedure stub

- RPCGEN produce gli stub per il server e il client da un insieme di costrutti descrittivi per tipi di dati e per le procedure remote in linguaggio RPC



SVILUPPO E SUPPORTO RPC

Nel caso di un utente che deve utilizzare le RPC, di varia tecnologia, abbiamo molte fasi, alcune facilitate e poco visibili, svolte tutte dal supporto, altre sotto il controllo utente

Dopo la specifica del contratto in IDL, …

FASI TIPICHE della IMPLEMENTAZIONE

- **compilazione di sorgenti e stub**
- **binding delle entità**
- **trasporto dei dati**
- **controllo della concorrenza**
- **supporto alla rappresentazione dei dati**

Alcuni ambienti facilitano altri meno, ma tutti hanno una qualche caratterizzazione specifica

FASI DI SUPPORTO RPC

compilazione di sorgenti e stub

- La compilazione produce gli stub che servono a semplificare il progetto applicativo e risponde alla necessità che cliente e servitore raggiungano un accordo sul servizio da richiedere / fornire

trasporto dei dati

- Il trasporto connesso e senza connessione è intrinseco allo strumento e tanto più veloce ed efficiente, tanto meglio (TCP vs. UDP)

controllo della concorrenza

- Il controllo consente di usare gli stessi strumenti per funzioni diverse, con maggiore asincronicità e maggiore complessità (ripetizione, condivisione di connessione, processo)

supporto alla rappresentazione dei dati

- per superare eterogeneità si trasformano i dati, tanto più veloce, tanto meglio, bilanciata con la ridondanza se ritenuta necessaria

RPC BINDING

Binding delle entità

- Il binding prevede come ottenere l'aggancio corretto tra i clienti e il server capace di fornire la operazione

Il binding del **cliente al servitore** secondo due possibili linee

Scelta pessimistica e statica

- La compilazione risolve ogni problema prima della esecuzione e forza un binding statico (nel distribuito) a costo limitato (ma poco flessibile)

Scelta ottimistica e dinamica

- Il binding dinamico ritarda la decisione alla necessità, ha costi maggiori, ma consente di dirigere le richieste sul gestore più scarico o presente in caso di sistema dinamico

BINDING **STATICO** vs. **DINAMICO**

fondamentale nella relazione tra cliente e servitore

BINDING DINAMICO RPC

Il **binding dinamico** tipico viene ottenuto distinguendo due fasi nella relazione cliente/servitore

- **Servizio (fase statica)** prima della esecuzione

  il cliente specifica a chi vuole essere connesso, con un nome unico identificativo del servizio (NAMING)

  in questa fase si associano dei nomi unici di sistema alle operazioni o alle interfacce astratte e si attua il binding con la interfaccia specifica di servizio
- **Indirizzamento (fase dinamica)** all'uso, durante la esecuzione

  il cliente deve essere realmente collegato al servitore che fornisce il servizio al momento della invocazione (ADDRESSING)

  in questa fase si cercano gli eventuali servitori pronti per il servizio (usando sistemi di nomi che sono stati introdotti proprio a tale scopo)

BINDING RPC

Parte di NAMING statica, SERVIZIO

- risolto con un numero associato staticamente alla interfaccia del servizio (nomi unici)

Parte di ADDRESSING dinamica, durante l'esecuzione

La parte dinamica deve avere costi limitati ed accettabili durante il servizio

1) ESPLICITA attuata dai processi

- Il cliente deve raggiungere un servitore
  si può risolvere con un **multicast o broadcast** attendendo solo la prima risposta e non le altre

2) IMPLICITA tramite un agente esterno, un servitore di nomi

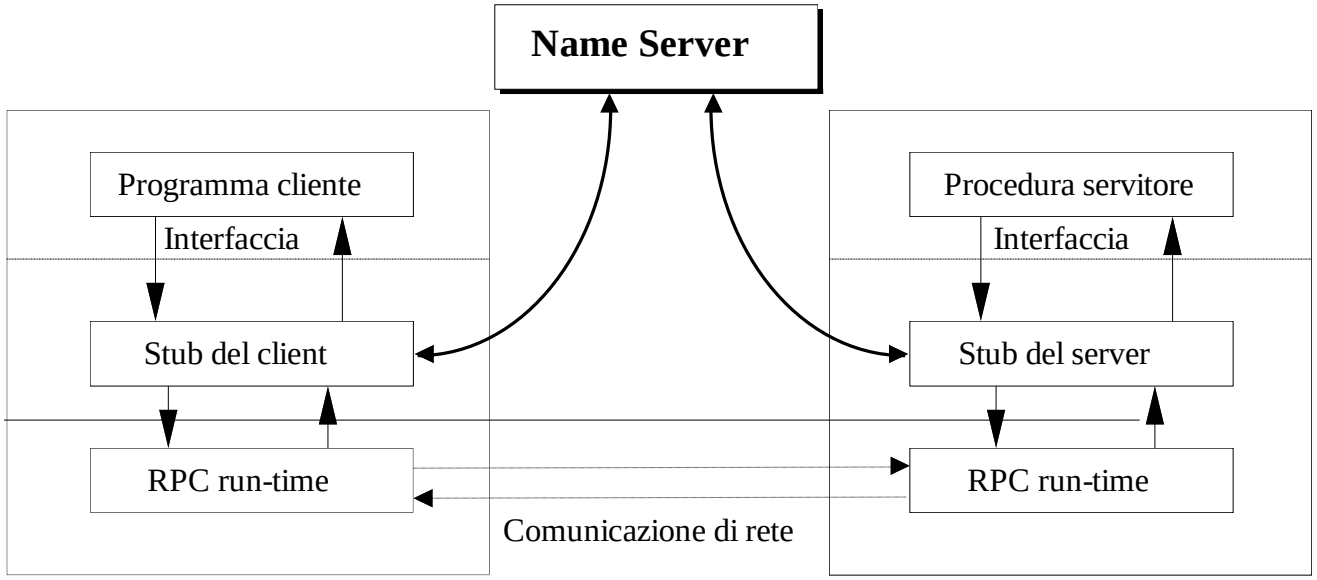
- uso di un name server che registra tutti i servitori e agisce su opportune tabelle di binding ossia di nomi, prevedendo funzioni di ricerca di nomi, registrazione, aggiornamento, eliminazione
- **SISTEMI di NOMI**

FREQUENZA DEL BINDING

In caso di BINDING DINAMICO

Ogni chiamata richiede un **collegamento dinamico**
Spesso dopo un primo legame si usa lo stesso binding ottenuto come se fosse statico per questioni di costo

- **il binding può avvenire meno frequentemente delle chiamate stesse**
- in genere, si usa lo stesso binding per molte richieste e chiamate allo stesso server



SISTEMI DI NOMI

Le RPC hanno portato a molti sistemi di nomi, detti **Binder**, **Broker**, **Name Server**, ecc, tutte entità di sistema per il binding dinamico

Un binder deve fornire operazioni per consentire agganci flessibili

Lookup	(servizio, versione, &servitore)	**funzione più usata**
Register	(servizio, versione, servitore)	
Unregister	(servizio, versione, servitore)	

Il nome del servitore (servitore) può essere dipendente dal nodo di residenza o meno. Se dipendente, allora ogni variazione deve essere comunicata al binder

Il BINDING attuato come **servizio coordinato di più servitori**

- Uso di binder multipli per limitare overhead e di cache ai singoli clienti o ai singoli nodi
- Inizialmente i clienti usano un broadcast per trovare il binder più conveniente

CONTROLLO E ASINCRONICITÀ

Necessità di RPC asincrone, o meglio **non bloccanti**

- il cliente non si blocca ad aspettare il servitore

Possibilità di modalità asincrone nei diversi ambienti di uso

- per maggiore parallelismo ottenibile
- con processi leggeri per il server

Problema fondamentale come ottenere il risultato.

Due punti di vista di progetto

RPC bassa latenza vs. **RPC alto throughput**

RPC bassa latenza

- tendono a mandare un messaggio di richiesta ed a trascurare il risultato - **realmente asincrone**

RPC throughput elevato

- tendono a differire l'invio delle richieste per raggrupparle in un unico messaggio di comunicazione

RPC ASINCRONE

Implementativamente, si usano sia supporti UDP o TCP, ottenendo semantiche diverse

Realmente asincrone (senza risultato)

Athena (XWindows)

- usa UDP e bassa latenza - semantica may-be

SUN

- usa TCP ad elevato throughput - semantica at-most-once
- invio di una serie di RPC asincrone e di una finale in batching

Asincrone (con restituzione di risultato)

Chorus - bassa latenza

- uso di una variabile che contiene il valore del risultato nel cliente

Mercury - alto throughput

- uso di stream per le richieste tenendo conto di azioni asincrone e sincrone

PROPRIETÀ DELLE RPC

Analisi delle proprietà di ogni sistema RPC

proprietà visibili all'utilizzatore

- **entità che si possono richiedere** operazioni o metodi di oggetti
- **semantica di comunicazione** maybe, at most once, at least once
- **modi di comunicazione** a/sincroni, sincroni non bloccanti
- **durata massima e eccezioni** ritrasmissioni e casi di errore

Proprietà trasparenti all'utilizzatore

Servitore sequenziale o parallelo ricercato via
uso di sistemi di nomi con broker unico centralizzato / broker multipli

presentazione dei dati linguaggio IDL ad hoc e generazione stub

passaggio dei parametri passaggio per **valore**, per riferimento

eventuali legami con le risorse del server

- persistenza della memoria per le RPC
- crescita delle operazioni via movimento di codice
- aggancio con il garbage collector

RPC DI SUN

RPC di SUN con visione a processi server sequenziali e non trasparenza allocazione

- operazioni richieste al nodo del servitore
- **entità che si possono richiedere** solo **operazioni o funzioni**
- **semantica di comunicazione** **at-most-once** e **at-least-once**
- **modi di comunicazione** **sincroni e asincroni**
- **durata massima** **timeout**
- **ricerca del servitore** **port mapper sul server**

Server sequenziale a default

Presentazione dei dati

- linguaggio IDL ad hoc XDR e generazione stub RPCGEN
- **Passaggio dei parametri** **passaggio per valore**

le strutture complesse e definite dall'utente sono linearizzate e ricostruite al server per essere distrutte al termine della operazione

- **Estensioni varie** broadcast, credenziali per sicurezza, ...

RMI DI JAVA

RMI visione per sistemi Java ad oggetti passivi con trasparenza alla allocazione (?), senza eterogeneità, e con scelte che non privilegiano la efficienza ma con parallelismo dei server

- **entità da richiedere** metodi di oggetti via interfacce
- **semantica di comunicazione** **at-most-once** (TCP)
- **modi di comunicazione** solo **sincroni**
- **durata massima e eccezioni** trattamento di casi di errore

Server Parallelo via thread

Ricerca del servitore uso di registry nel sistema broker unico centrale non sono forniti broker multipli (distribuiti?) ma si possono anche avere organizzazioni più complesse

- **presentazione dei dati** generazione **stub e skeleton**
- **passaggio dei parametri** **passaggio a default per valore**, passaggio per **riferimento di oggetti con interfacce remotizzabili**

eventuali legami con le risorse del server e aggancio con il sistema di sicurezza, e aggancio con il garbage collector