

Client/Server Computing, Middleware, RPC

Domanda: è possibile scrivere un'applicazione che viene supportata da due SO eterogenei?

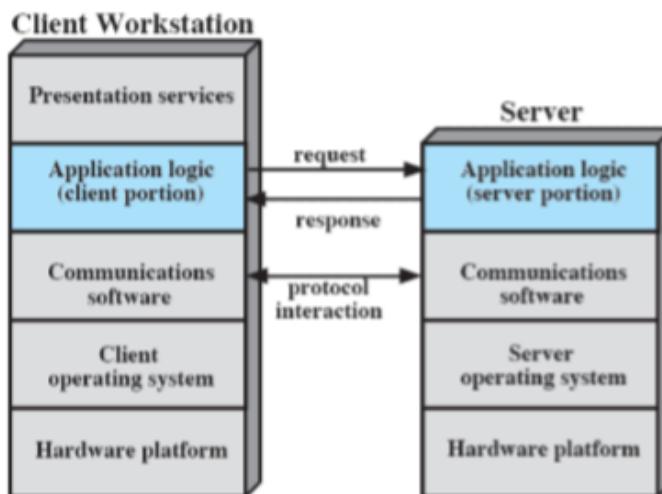
È importante notare che la loro eterogeneità si riversa anche su le Application Programming Interface (API). La risposta è sì, grazie all'uso di uno stadio intermedio che copre l'eterogeneità → MIDDLEWARE (M).

Ne esistono diversi tipi; il più semplice è il Client/Server, i più complessi permettono di lavorare su più piattaforme.

È bene notare, però, che la linea di demarcazione tra SO e M non è fissa → i SO sono collezioni di routine che evolvono con le applicazioni, cioè se un meccanismo diventa standard viene direttamente inglobato es. RPC

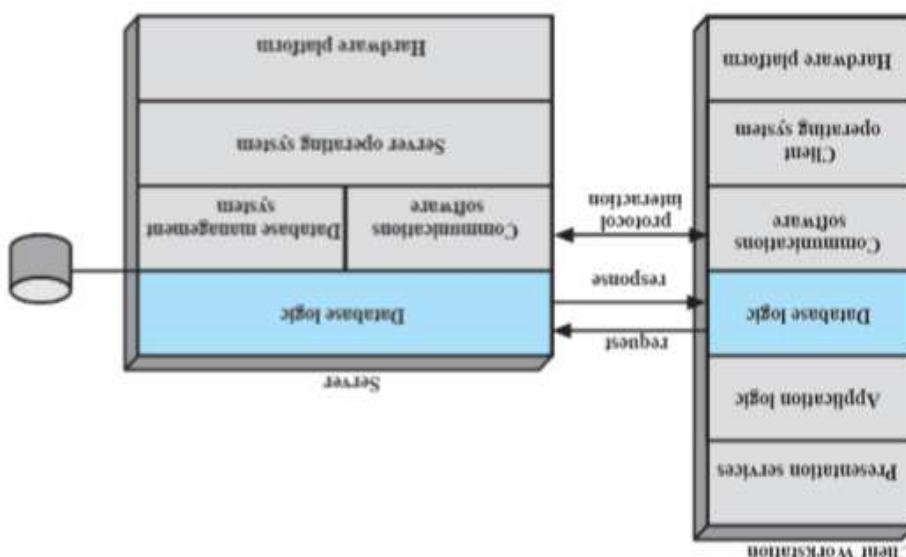
Computazione Client server

- ◊ Server macchina importante → fa tutto o la maggior parte del lavoro → serve per condividere informazioni es. DATABASE
- ◊ Structured Query Language (SQL) → linguaggio sviluppato per l'interazione con database
- ◊ I client richiedono informazioni o servizi ai server
- ◊ La loro struttura viene divisa in 2 livelli: APPLICAZIONE e COMUNICAZIONE
- ◊ Essi possono avere piattaforme e SO diversi, ma queste diversità dei livelli inferiori spariscono se condividono uno stesso protocollo di comunicazione e supportano la stessa applicazione
- ◊ La parte applicativa si suddivide in 3 sottolivelli:
 1. DATABASE LOGIC (nel caso di database) → serve per mantenere tutte le proprietà del database
 2. APPLICATION LOGIC → visita diversi database per cercare le informazioni
 3. PRESENTATION SERVICES → presentazione dei dati all'utente
- ◊ Generica architettura:



In questa struttura alcune applicazioni sw girano sul server e altre sul client

- ◊ Occupiamoci del caso specifico in cui un server è un database (ma vale anche in generale):
 - L'interazione tra client e server avviene tramite transazioni (il primo fa una richiesta a cui il secondo risponde)
 - Il server è responsabile per il mantenimento del database



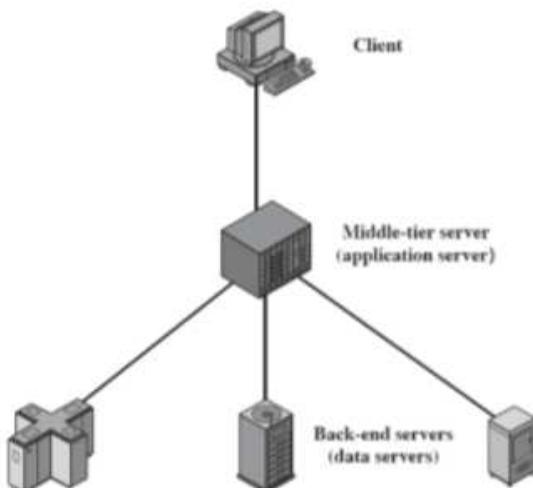
Organizzazione:

1. HOST-BASED PROCESSING → tutto il livello di applicazione è contenuto nel server → il client è vuoto (definito THIN-CLIENT) → è un terminale remoto (es. Remote desktop application)
2. SERVER-BASED PROCESSING → il sottolivello PRESENTATION LOGIC (il più alto) viene eseguito dal client → il server si occupa di tutta la computazione, ma il client provvede all'interfaccia grafica per la presentazione dei dati (es. google → il completamento della barra di ricerca è fatto dalla nostra macchina)
3. CLIENT-BASED PROCESSING → via via la potenza computazionale del client aumenta (come il suo peso), fino ad arrivare alla condivisione del sottolivello APPLICATION LOGIC
4. COOPERATIVE PROCESSING → questa volta il sottolivello condiviso è il DATABASE LOGIC → in quest'ultimo caso vi è un aumento dei costi soprattutto riguardo la consistenza dei dati

Dal punto 3 e 4 si capisce che i due terminali devono comunicare, tramite un sistema più potente rispetto al classico tcp/ip in quanto vi è un trasferimento di dati maggiori → vengono introdotti i motori publish/subscribe che si trovano all'interno di grandi sistemi come twitter o facebook → essi rientrano nel concetto di M

È stata pensata un'altra architettura basata su 3 livelli: THREE-TIER C/S ARCHITECTURE

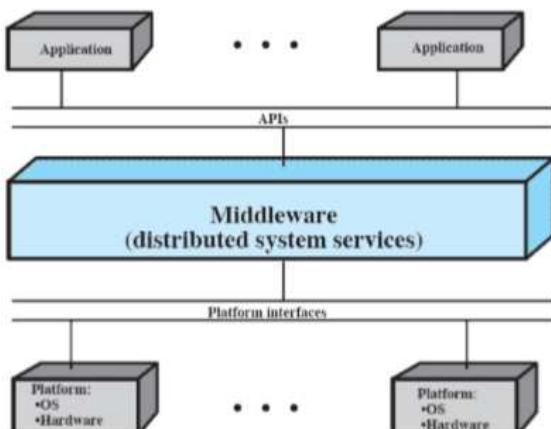
1. Client → è una User Machine che solitamente è molto leggera (THIN-CLIENT)
2. Middle-tier → funge da ponte: converte i protocolli e unifica/integra i dati ricevuti dalle diverse sorgenti → in più in caso di grosso carico divide le varie query nelle diverse copie di database
3. Backend Server



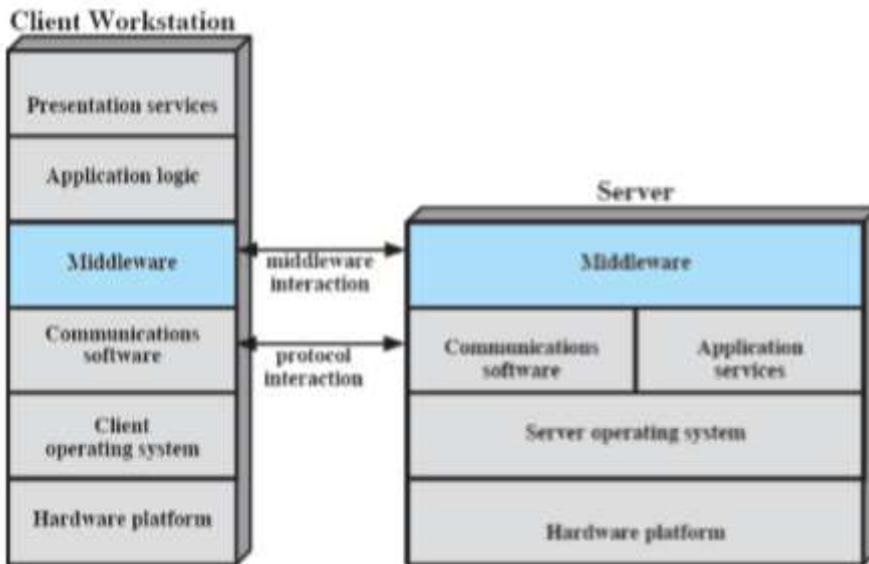
Middleware

Il M è un sw di connessione che consiste di un insieme di servizi e/o di ambienti di sviluppo di applicazioni distribuite che permettono a più entità (processi, oggetti, ...) residenti su uno o più elaboratori, di interagire attraverso una rete di interconnessione a dispetto di differenze nei protocolli di comunicazione, architetture dei sistemi locali, SO, ...

Quindi è un set di strumenti che forniscono un mezzo uniforme alle risorse di un sistema su tutte le piattaforme. Permette ai programmati di utilizzare lo stesso modo per accedere ai dati (attraverso APIs uguali per tutti).



M è il collante tra le piattaforme inferiori e l'applicazione. Con le prime si collega tramite le PLATFORM INTERFACES, invece con la seconda grazie alle APIs.



M, come detto, è un insieme di servizi; essi sono:

- ◊ Servizio di Comunicazione → necessaria per nascondere la disomogeneità legata alla rappresentazione dei dati usata dai vari elaboratori, ai SO locali e alle differenti reti che costituiscono l'infrastruttura della piattaforma → es. RPC, messaggistica applicativa, modello PUBLISH/SUBSCRIBE...
- ◊ Servizi di astrazione e cooperazione → cuore del M → tra i servizi che comprendono ci sono:
 - Directory Service → identifica e localizza un entità → rende le applicazioni al di sopra di M indipendenti dai sottosistema di instradamento dei messaggi
 - Security Service → protegge gli elementi critici attraverso tecniche come l'autenticazione, l'autorizzazione o la crittografia
 - Time Service → assicura che tutti i clock interni tra client e server siano sincronizzati → fondamentale per la comunicazione
- ◊ Servizi per le applicazioni → permette all'applicazione di avere accesso diretto con la rete o con i servizi offerti da M
- ◊ Servizi di amministrazione del sistema
- ◊ Ambiente di sviluppo applicativo

L'obiettivo principale del M è l'INTEROPERABILITA' e la CONNETTIVITA' per applicazioni distribuite su piattaforme eterogenee. Quindi tra diversi sistemi sia con grandi (informativi) che con piccole dimensioni.

La necessità dei M è nata quando le aziende, con sistemi informativi diversi, iniziarono a fondere. Quindi c'era necessità di un modo per farle cooperare e comunicare facilmente.

La problematica principale è capire la forma giusta del M. Infatti, in molti casi, offre più servizi di quelli necessari → problema progettuale e di prestazioni

La classe più gettonata di M è il MESSAGE-ORIENTED M che si basa su scambi di messaggi in modo asincrono (il server non deve per forza essere in funzione).

La trasmissione e la ricezione dei messaggi scandiscono l'evoluzione dell'applicazione.

Ne esistono vari tipi:

1. Code di messaggi → i messaggi inviati dal client ad un server vengono immagazzinati in una coda fino a che il ricevente non li preleva → permette di usare un politica di priorità, di bilanciare il carico, di gestire delle applicazioni tolleranti ai guasti e di migliorare le prestazioni (client non-bloccante)
2. Publish/Subscribe → la comunicazione non è punto – punto → viene usata un'architettura message bus → la comunicazione avviene tramite eventi catalogati → entrano in gioco due entità: il Publisher che invia/pubblica i messaggi e i Subscriber che sono i riceventi → i publisher pubblicano i messaggi con un determinato SUBJECT, essi possono non conoscere i loro subscriber → un processo può essere publisher di alcuni soggetti e subscriber di altri → i soggetti formano uno spazio unico e gerarchico → il sw che implementa tale paradigma deve controllare di non inondare la rete con copie dello stesso messaggio → se il subscriber è attivo viene fatta una upcall, altrimenti viene immagazzinato il messaggio all'interno di un elemento architetturale

Le Remote Procedure Calls (RPCs) sono un altro tipo di middleware, forse i più importanti.

Esse si basano sul meccanismo di chiamata a procedura → procedure chiamate da una principale e durante questa interazione avviene un passaggio di parametri.

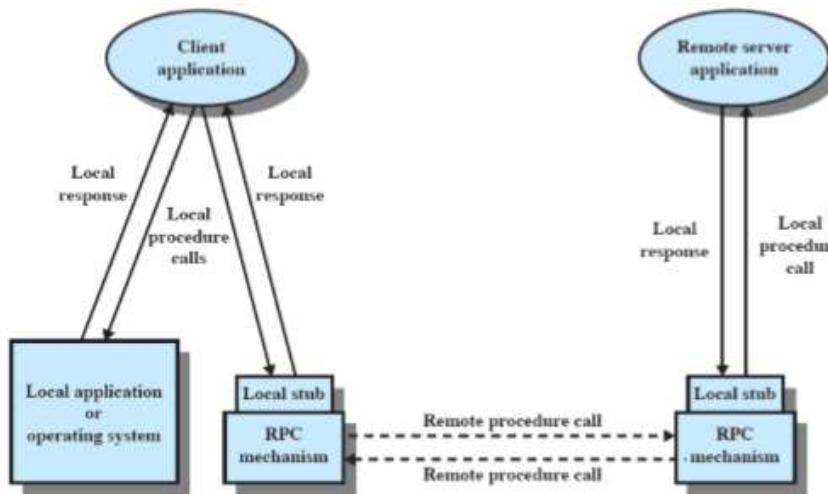
Tutto ciò, però, comporta dei problemi. Ad esempio: la comunicazione avviene tra due terminali diversi, quindi con due memorie diverse; quando viene passato un puntatore, esso, probabilmente, non ha senso sull'altro host, in quanto può essere implementato un immagazzinamento diverso.

Negli anni '80 i server erano molto pesanti, a differenza dei client. I primi conservavano i file system che erano messi al servizio di tutti. Per far comunicare facilmente chiamante e chiamato, il sistema delle chiamate di procedure fu estesa su rete. Un programmatore, però, deve essere consapevole delle limitazioni che posso nascere e capire bene il loro funzionamento.

Obiettivo fondamentale è di rendere le RPCs il più possibile simili alle chiamate locali.

Ci fu una larga accettazione → necessaria fu la standardizzazione

Funzionamento: abbiamo un chiamante ed un chiamato → il client invia un messaggio REQUEST che viene elaborato dal server (intanto il client è bloccato). Infine quest'ultimo invia un messaggio di REPLY.



Il passaggio di parametri avviene attraverso la rete, che può essere paragonato ad un tubo in cui riescono ad entrare solo bit. Quindi è importante anche focalizzarsi sulla compressione dei dati. Infatti la modalità con cui viene fatta deve essere condivisa dai due end-point. Ciò è fondamentale soprattutto se vengono inviate strutture complesse. Esempio: invio una matrice, è importante che il server non legga le righe come colonne.

Differenze con chiamate locali:

1. All'interno di un sistema centralizzato, quando chiamiamo una procedura, essa viene eseguita secondo certi parametri di input che poi producono un determinato output → si ha la certezza che l'esecuzione avverrà una sola volta. Nelle RPCs non sempre si ha questo funzionamento
2. Le RPCs sono legate alle applicazioni di cui fanno parte (Application Specific) → l'applicazione si trova sia sul client che su server. Nelle chiamate locali, invece, non vi è specificità
3. Differenza principale è legata ai MESSAGGI: essi, muovendosi attraverso un sistema inaffidabile, possono perdersi
4. In una macchina centralizzata se si rompe tutto, le chiamate si bloccano. Invece se il server si rompesse, il client potrebbe non capire se si è rotto l'interlocutore o si sono persi i messaggi

Problemi legati alle RPCs:

- Perdita e ordinamento dei messaggi
- Rottura del server
- Invio di risposte sbagliate da parte del server → sia in modo naturale (guasto) sia innaturale (intruso)

Il principale modo per risolverli, è creare un'opportuna struttura dei server e dei client, ampliando la parte sw.

Quindi una Chiamata a Procedure Remota (RPC) trasforma l'interazione Client/Server in una chiamata a procedura, simile a quella locale, nascondendo, però, al programmatore la maggiore parte dei meccanismi implementativi che la compongono, come:

- ◊ L'interscambio di messaggi (ad esempio quelli di sincronizzazione o il passaggio di parametri)
- ◊ La localizzazione del server che fornisce il servizio (legato anche al problema del guasto del server → un name server gestisce una tabella di corrispondenze tra la procedura chiamata e gli ip dei possibili server destinatari, così da non costringere il client a far riferimento ogni volta allo stesso=

- ◊ Le possibili differenti rappresentazione dei dati delle macchine coinvolte nell'interazione
- Questi dettagli vengono nascosti in 4 fasi:

1. A tempo di scrittura del codice → le RPCs, usate e fornite, devono essere dichiarate esplicitamente dal programmatore, attraverso l'importazione e l'esportazione delle definizioni delle interfacce
2. A tempo di esecuzione del codice → in ogni macchina dovrà avere un supporto a tempo di esecuzione per le RPC (RPC run-time support) → chiamato anche demone → è un processo che capisce quali sono i messaggi legati ad una chiamata remota → il suo compito nel server è passare i giusti parametri alla RPC collegata, invece, nel lato client è di trovare il server appropriato (localizzazione)
3. A tempo di compilazione → per ogni chiamata remota, vengono aggiunte delle linee di codice al programma originario (stub) che agevolano il passaggio dei messaggi e parametri (modalità di compressione condivisa) e l'inserimento delle giuste chiamate al run-time support → viene creato un unico eseguibile
4. A tempo di collegamento

I meccanismi fondamentali sono due:

- Un protocollo che nasconde le insidie della rete (perdita di pacchetti e riordinamento dei messaggi)
- Un meccanismo per impacchettare gli argomenti dal lato chiamante e per spacchettarli dal lato chiamato

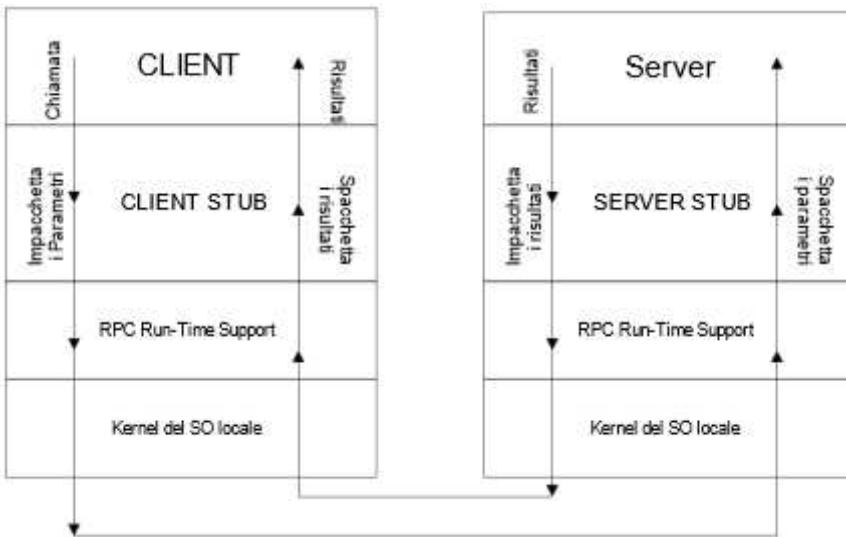
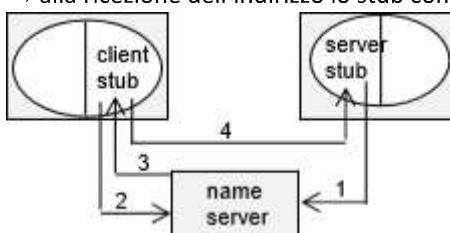


Foto: Interazione a Run-Time → Il client fa partire la chiamata → nella fase di compilazione aggiunge lo STUB, che impacchetta i parametri in maniera standard → essi vengono passati al run-time support, il quale, se necessario, localizza l'ip del server → il supporto usa le primitive di socket, le quali rientrano nel kernel del SO locale, per inviare i pacchetti → I PACCHETTI VIAGGIANO (viene usato UDP come protocollo di comunicazione) → arrivo al server → il run-time support li porta nel socket appropriato → il server li spacchetta e li attribuisce ad una determinata procedura, prima di passarglieli li rende leggibili → la risposta viene impacchettata attraverso il server stub → attraversa tutto il percorso al contrario

Localizzazione del server: avviene in tre modi fondamentali 49

1. Metodo Statico: all'interno del client viene cablato l'indirizzo IP del server → meno flessibile → se il server è guasto arriva un errore
2. Metodo Dinamico: viene utilizzato un protocollo simile ad Arp (che cercava la corrispondenza tra MAC e IP) → lo stub del client invia una richiesta broadcast in modo da sapere l'IP del server in grado di eseguire la RPC desiderata → risponde il supporto run-time delle RPC di ogni macchina che implementa il servizio richiesto
3. Name Server: metodo più evoluto → è un host fisico → aggiorna una tabella in cui riporta le corrispondenze tra procedura e lista di server che la eseguono (deriva dai DNS che scambiano gli indirizzi mnemonici con gli indirizzi IP) → Funzionamento: il server stub notifica che viene eseguita una nuova procedura a quell'indirizzo ip (operazione che avviene in fase di compilazione) (1) → il client chiede al name server se esiste una determinata procedura (2) → esso risponde con l'IP address del server giusto o con un errore (3) → alla ricezione dell'indirizzo lo stub completa l'impacchettamento ed invia il messaggio (4).



Problema semantica delle chiamate: All'interno di una procedura classica noi potremmo usare le seguenti chiamate:

Call by Reference → viene passato un puntatore il riferimento non ha senso in un altro elaboratore

Call by Copy/Restore (by value) → passaggio di valori

Però, queste, nei sistemi distribuiti, creano dei problemi. Infatti il primo tipo non ha senso, in quanto si fa riferimento a memorie differenti.

Il secondo, invece, porta ad un side-effect dei valori passati. Mostriamo il seguente esempio:

CLIENT SIDE	SERVER SIDE
begin	procedure doppioincr (var x,y: integer)
.....
a=0;	x:= x+2;
doppioincr(a,a);	y:= y+3;
writeln (a); ...	
end	end

Il client chiama una procedura passando lo stesso parametro, quindi lo stesso riferimento:

- Se la chiamata avvenisse nello stesso calcolatore, le variabili x e y sarebbero tutte e due a, quindi x=x+2 sarebbe a=a+2=0+2=2 e y=y+3 sarebbe a=a+3=2+3=5 quindi ciò che ritorna è 5
- Nel caso di una RPC, il pacchetto REQUEST (con la variabile a) viene letta dal server come due parametri distinti e quindi x=a=0 e y=a=0. Il messaggio REPLY conterrà due risultati 2 e 3. Quando il client riceve la risposta, per effetto della linearizzazione del canale di rete e per il fatto che ci troviamo su macchine sequenziali, assegna ad a il valore 2 o 3, ciò dipende dall'implementazione dello stub del client.

Questo, quindi, porta a degli errori. Purtroppo non possono essere evitati, ma controllati attraverso l'attenzione del programmatore, ad esempio non inviando lo stesso parametro quando ne sono richiesti due.

Un'altra problematica è la SEMANTICA delle RPC:

In un sistema centralizzato abbiamo la certezza che una chiamata di procedura avvenga un'unica volta, cosa non certa, invece in una comunicazione distribuita.

Ad esempio:

Client invia richiesta → Server esegue la computazione → la risposta si perde → il time-out del client scade e quindi rinvia la chiamata → IL SERVER RIESEGUE LA COMPUTAZIONE → arriva la risposta

Quindi la semantica di una RPC può essere:

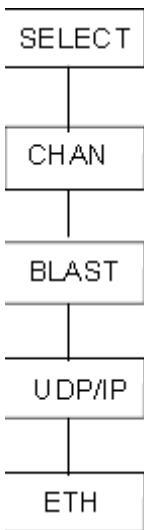
- AT LEAST ONE → esempio → se lo stub del client esegue k volte la richiesta della RPC, nel peggior dei casi, il server ricomputerà la risposta esattamente k volte
- AT MOST ONCE → al più la richiesta viene rieseguita una volta → infatti, se è già stata inviata ritorna un codice errore
- EXACTLY ONE → è il comportamento ideale in quanto emula una chiamata locale, però è anche molto complesso eseguirlo, infatti devono essere implementate delle modifiche:
 - LATO SERVER: il terminale deve capire che ha già ricevuto una determinata richiesta, e quindi deve solo rinviare i risultati → perciò, nel server viene introdotto un disco dove sono immagazzinati tutti i risultati delle RPC (Logging)
 - LATO CLIENT → tutte le richieste devono essere numerate (SEQUENCE NUMBER) → il client deve tenere conto di un numero di REINCARNAZIONE (incrementato di 1 ad ogni restart) → infatti se muore, dopo aver già fatto una RPC, alla sua riaccensione rinvia la richiesta con lo stesso numero di sequenza, ciò comporterebbe ad una ricomputazione del server, cosa evitata grazie al confronto con il numero di reincarnazione memorizzato

Sottosistema di Comunicazione

Il protocollo usato a livello di trasporto, per inviare una RPC, è UDP. Poiché non c'è nessuna garanzia di consegna, per gestire l'invio affidabile dei dati, si fa riferimento ad un protocollo superiore. Questo può usare due tipi di tecniche di gestione della spedizione dei dati:

1. STOP AND WAIT
2. BLAST

Facciamo riferimento alla seguente catena protocollare. Essa appartiene a SunRPC:



Select, Chan e Blast sono protocolli che mantengono la semantica (esempio: At Least One) ed aumentano le probabilità che non si perdano dei pacchetti. Essi costituiscono l'RPC Protocol. Fanno distinzione tra sender e receiver ma non tra client e server.

Blust:

Esso divide (e riassembra) le informazioni in frammenti e poi tramite UDP li spedisce. Quindi il suo compito è di gestire i riscontri.

Caratteristiche:

1. Supporta una ritrasmissione selettiva segnalata dalle SRR
2. Vengono gestiti dei riscontri parziali
3. sono usati 3 timer: Done, Last_Frag e Retry

Vediamo in dettaglio il funzionamento dei timer:

LATO SENDER:

- Blast invia tutti i frammenti in maniera sequenziale e setta il timer DONE
- Se riceve una SRR, invia i frammenti mancanti e resetta il timer
- Se riceve una SRR speciale, cioè "tutti i frammenti sono stati ricevuti", libera la memoria
- Se il timer Done scade, rinuncia quindi libera la memoria ed avvisa il protocollo superiore (CHAN)

LATO RECEIVER:

- Appena riceve il primo frammento setta il timer LAST_FRAG (calcola il tempo massimo necessario per ricevere tutti i dati che dovrebbero arrivare)
- Se riceve tutti i dati, li riassembra e li passa al livello superiore
- Altrimenti potrebbero aprirsi 4 scenari:
 1. L'ultimo frammento arriva ma il messaggio non è completo → invia SRR e setta il timer RETRY
 2. Last_Frag scade → invia SRR e setta Retry
 3. Retry scade (per la prima o seconda volta) → invia SRR e setta Retry
 4. Retry scade per la terza volta → rinuncia e libera la memoria occupata dagli altri frammenti

Parte potrebbe essere parametri → richiamato fino a che tutti i parametri non sono trasferiti

Performance:

Quando la rete è NICE cioè ottimale, la lunghezza dei timer non è così fondamentale:

Done, solitamente, è un valore abbastanza grande; invece, Retry è più corto ma non deve esserlo troppo, altrimenti si rischia il rinvio di troppi SRR.

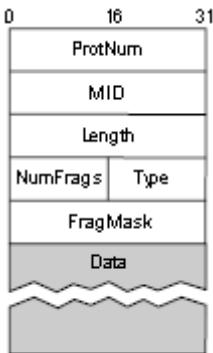
Conclusione

Blast è un protocollo persistente (testardo → Stubborn), cioè continua a richiedere la trasmissione dei frammenti mancanti. Questo lo rende utile soprattutto quando vengono scambiati messaggi di grandi dimensione (parte potrebbero essere parametri).

Esso non si occupa di garantire l'invio del messaggio completo (lavoro di Chan) ma solo della trasmissione dei pacchetti a lui passati. Quindi esso non è in grado di spedire il messaggio completo; questa scelta è legata alle

prestazioni, infatti se dovesse attendere il rinvio di tutti i frammenti (perciò non solo quelli richiesti) ci sarebbe una sostanziale perdita di velocità.

Formato



MID → intestazione: tutti i frammenti hanno la stessa

NumFrgs → numero del frammento

FragMask → distinzione tra Data o SRR

Essendo al massimo 32 bit, Blast può inviare al massimo 32 pacchetti per messaggio.

Chan:

Si occupa di gestire le richieste e le repliche, quindi opera con i messaggi applicativi, cioè semantici, in cui si trovano le informazioni da scambiare. Suo compito è anche di richiamare blust fino a che tutto non è stato inviato. Da ciò si deduce che garantisce l'invio dei dati. Si occupa anche del riassemblaggio dei pacchetti. Sincronizza anche il client e server (differenza con blust → ci troviamo ad un livello superiore).

Chan tiene conto della perdita di messaggi, quindi ognuno di essi è memorizzato fino a che non arriva l'ACK relativo. Se quest'ultimo non dovesse arrivare, allo scadere del time-out RETRANSMIT, settato in precedenza, viene inviato nuovamente l'intero messaggio. Ciò comporta, però, un lavoro nel lato del destinatario, infatti, tramite il campo MID, dovrà accorgersi che i frammenti arrivati sono un duplicato. Da tutto questo discorso si evince che viene usata una semantica AT LEAST ONE (sviluppato sui sistemi UNIX).

Per avere una semantica EXACTLY ONE, è necessario del sw accessorio, che permette di non avere le problematiche viste prima, implementato al di fuori dei protocolli trattati.

Differenza con TCP

- Funzionamento TCP: caso perdita del riscontro → invio richiesta e settaggio di un timer T → se non arriva l'ACK e scade T → rinvio pacchetto e setto a 2 volte T → se non arriva risposta e scade di nuovo T → rispedisco il messaggio e aggiorno T a 4 volte → fino a che non viene ricevuto un riscontro, T viene impostato esponenzialmente (in verità per default fino a 16)
Funzionamento Blast: caso perdita del riscontro → riprova in maniera testarda fino a che non riesce ad ottenere un Ack, se molla, torna da Chan che lo richiama per riprovare l'invio
- Funzionamento TCP: caso senza perdite → invio pacchetto e settaggio T → ricezione del riscontro → invio di 2 pacchetti → ricezione Ack → spedizione di 4 pacchetti →... Utilizzo della banda esponenziale → comportamento SLOW-START
Funzionamento Blast: caso senza perdite → invio di 32 pacchetti → Selective Retransmission (SRR – meccanismo che permette di evitare la spedizione di numerosi Ack da parte del receiver) → rinvio dei pacchetti richiesti → Ack → spedizione di 32 pacchetti → ... Utilizzo, fin da subito, di tutta la banda
- TCP è nato negli anni '70, quando la rete non era ottimale, e si cercava di risparmiare il più possibile risorse. Le RPC, invece, nascono circa 10 anni dopo e sono state create per utilizzare tutta la banda
- TCP, nel caso di perdita di pacchetti, porta ad un aumento del ritardo
Blast, permette di risparmiare se tutto va bene, ma se la rete non funziona, comporta un notevole danno
- Blast è molto efficiente quando devono essere inviati messaggi di grandi dimensioni
N.B. Ciò che ci permette, in meccanismi peer-to-peer (che si appoggiano a TCP), di non dover ricominciare la comunicazione dall'inizio è il software accessorio, esempio quando si scarica un film e cade la connessione.

Select

È il dispatcher, cioè si occupa di assegnare alla giusta RPC i dati che riceve. Per distinguere le varie chiamate remote, esistono 2 modi:

1. FLAT → identificatore unico per ogni procedura
2. Gerarchico → dipende dal main che contiene le RPC → identificatore del programma + numero sequenziale di procedura

Formattazione dei dati

Come sappiamo, chiamante e chiamato si trovano su due macchine distinte, quindi con processori ed architetture diverse, questo comporta a dover regolamentare il passaggio dei parametri. Infatti deve essere un STREAM STRUTTRATO.

La regolamentazione avviene in una codifica e decodifica comune delle informazioni passate.

Le difficoltà principali sono legati alla rappresentazione di due tipi di dato:

1. INTERI → il problema è legato a come sono organizzati in memoria → LITTLE-ENDIAN o BIG-ENDIAN o quanti byte sono concessi (1,2,4)
2. FLOATING POINT → la loro raffigurazione può essere STANDARD (IEEE 754) o non standard → legato al numero di bit concessi per la virgola mobile

Queste complicazioni, quindi, si riducono ad un problema di interoperabilità. Il modo per superarla è l'utilizzo di FORME CANONICHE. Esse sono una sorta di lingua intermedia: A codifica i dati secondo una forma, scelta da tutti e due → B riceve i dati e li decodifica in modo da adattarli per la sua organizzazione.

Si potrebbe utilizzare la tecnica Receiver-Makes-Right → non viene accordata precedentemente una forma, ma è il ricevitore che si occupa di capire la forma utilizzata, confrontandole con quelle che possiede. Il problema che ne consegue, è che potrebbe non conoscere quella usata e quindi, non si potrebbe comunicare.

I dati, all'interno di una forma, possono essere:

- Tagged → nel pacchetto viene specificato: il tipo, la lunghezza e il valore (TYPE, LEN, VALUE)
- Untagged → non c'è una struttura del pacchetto prestabilita

Alcune forme canoniche:

EXTERNAL DATA REPRESENTATION (XDR) → più famosa, usata per le prime RPC (SunRPC), i dati sono UNTUGGED (tranne per le strutture dinamiche, array, di cui deve conoscere la dimensione)
 ABSTRACT SYNTAX NOTATION ONE (ASN) → è un ISO standard cioè usato per comunicazione standardizzata (esempio: protocolli Simple Network Management Protocol → controllano in remoto macchine o elementi legati alla rete, switch, router...), i dati sono TAGGED

Esistono altri tipi di forme canoniche, non ne esiste una vincente → dipende dall'applicazione

Una forma che utilizza il meccanismo Receiver-makes-right è il NETWORK DATA REPRESENTATION → la prima cosa passata è la rappresentazione interna dell'elaboratore (fig.), in modo che il ricevente sa se hanno la stessa architettura o deve trasformare i parametri.

0	4	8	16	24	31
IntegrRep	CharRep	FloatRep	Extension 1	Extension 2	

- **IntegerRep**
 - 0 = big-endian
 - 1 = little-endian
- **CharRep**
 - 0 = ASCII
 - 1 = EBCDIC
- **FloatRep**
 - 0 = IEEE 754
 - 1 = VAX
 - 2 = Cray
 - 3 = IBM