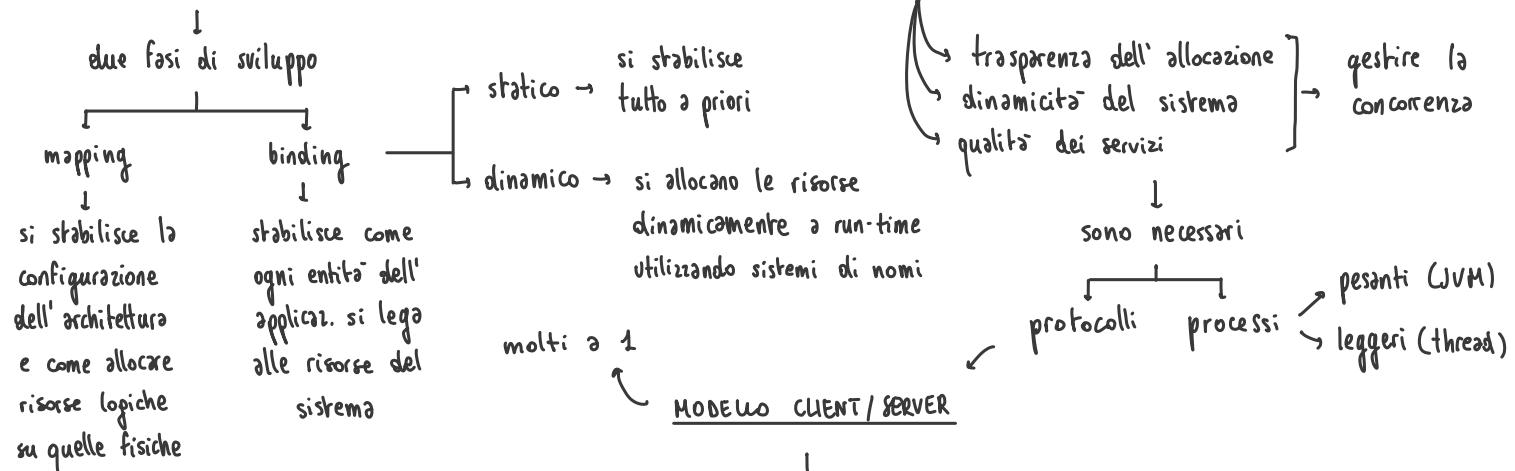


SISTEMI DISTRIBUITI → comunicano per ottenere risultati coordinati → accesso a risorse remote



TIPI DI SERVER

stateless stateful

- + leggero e affidabile
- client però deve mantenere lo stato

gerarchia (albero) di nomi logici e binding dinamico

agente detto name resolver posseduto da ogni dominio usa memoria cache x memorizzare op. precedenti

efficiente e + velocità di risposta

per accedere ai servizi è necessario poterli identificare e trovare

DNS ← sistemi di nomi

servizio di nomi basato su un insieme di gestori coordinati che si organizzano per rispondere a query che richiedono l'ip

→ i gestori gestiscono le coppie {NL, NF?} attuando la corrispondenza tra loro, al client basta il nome logico (dominio)

2 tipi di query

iterativa ricorsiva

sequenza di request/reply res. risposta o altro DNS

catena di server r/r che collaborano per fornire una risposta

sequenziale → richieste 1 alla volta, le altre in coda → basso utilizzo delle risorse

simile a modello ad agenti (molti server forniscono un servizio unico)

per accedere ai servizi è necessario poterli identificare e trovare

DNS ← sistemi di nomi → server capaci di fornire servizi di gestione e mantenimento dei nomi

nome logico del server trasparente con binding statico oppure dinamico

gestori partizionati gestori replicati

ciascuno responsabile di una sola parte

ciascuno responsabile insieme ad altri di una parte dei riferimenti

SOCKET JAVA → end-point di un canale di comunicazione bidirezionale → package `java.net` → processo identificato da {IP, porta}

con connessione

STREAM ↗ TCP → affidabile
bidirezionale
at-most-once

la connessione tra i processi C/S è definita dalla quadrupla univoca {IP1, porta1, IP2, porta2} e dal protocollo TCP, con due tipi di socket distinte

client ↘ ↗ server

`java.net.Socket`

`java.net.ServerSocket`

identificazione del processo senza pid → doppio sistema di nomi → messaggi consegnati alla porta, non direttamente al processo ↗ socket legano il processo ad un nome globale

senza connessione

↓

MULTICAST

costruttore

`MulticastSocket`

metodi:

`joinGroup(inet)`
`leaveGroup(inet)`

esiste un solo tipo di socket datagram sia per client che per server

↓

package `java.net.DatagramSocket`

permette di creare una socket connessa, i costruttori creano la socket e la legano ad una porta locale, per poi connetterla alla porta remota

sia costruttori che prendono porta e `inetAddress` remoti sia remoti + locali (`IOException`)

in attesa di richieste il server si blocca, si sblocca quando riceve

per lettura e scrittura

basso livello alto livello

`public InputStream
getInputStream()`

`DataInputStream
DataOutputStream`

`public OutputStream
getOutputStream()`

`writeUTF, writeln
readUTF, readInt`

restituiscono stream di byte (senza nessuna formattazione dei mess.) che incapsula il canale di comunicazione

`ShutdownInput()`

CLOSURA

`public synchronized void close()`

`IN` ↗
elimina subito

`OUT` ↗
tiene i dati x un po'

socket

multicast

↓

DATAGRAM

permette unicamente di accettare richieste di connessione provenienti dai vari client

due costruttori, o solo porta o porta + dim. della coda delle richieste (5 default)

il server si deve mettere in attesa di nuove richieste di connessione tramite la primitiva `accept()`

blocca il server fino all'arrivo di una richiesta e poi restituisce un oggetto della classe `Socket`

la chiamata di `accept` è sospensiva

se il server è parallelo, all'accettazione può generare un thread che eredita la connessione e la chiude al termine (poi server principale torna in attesa)

↓

OPZIONI → interfaccia `SocketOptions`

`SetSoLinger(bool on, int linger)`

imposta dopo quanto tempo (intervallo di linger) vengono scartati i pacchetti in output

`SetTcpNoDelay(bool on)`

`SetKeepAlive(bool on)`

↓

OPZIONI

visto che la receive è sincrona bloccante

oppure x la dim. del buffer

↓

`SetSoTimeout(int)`

↓

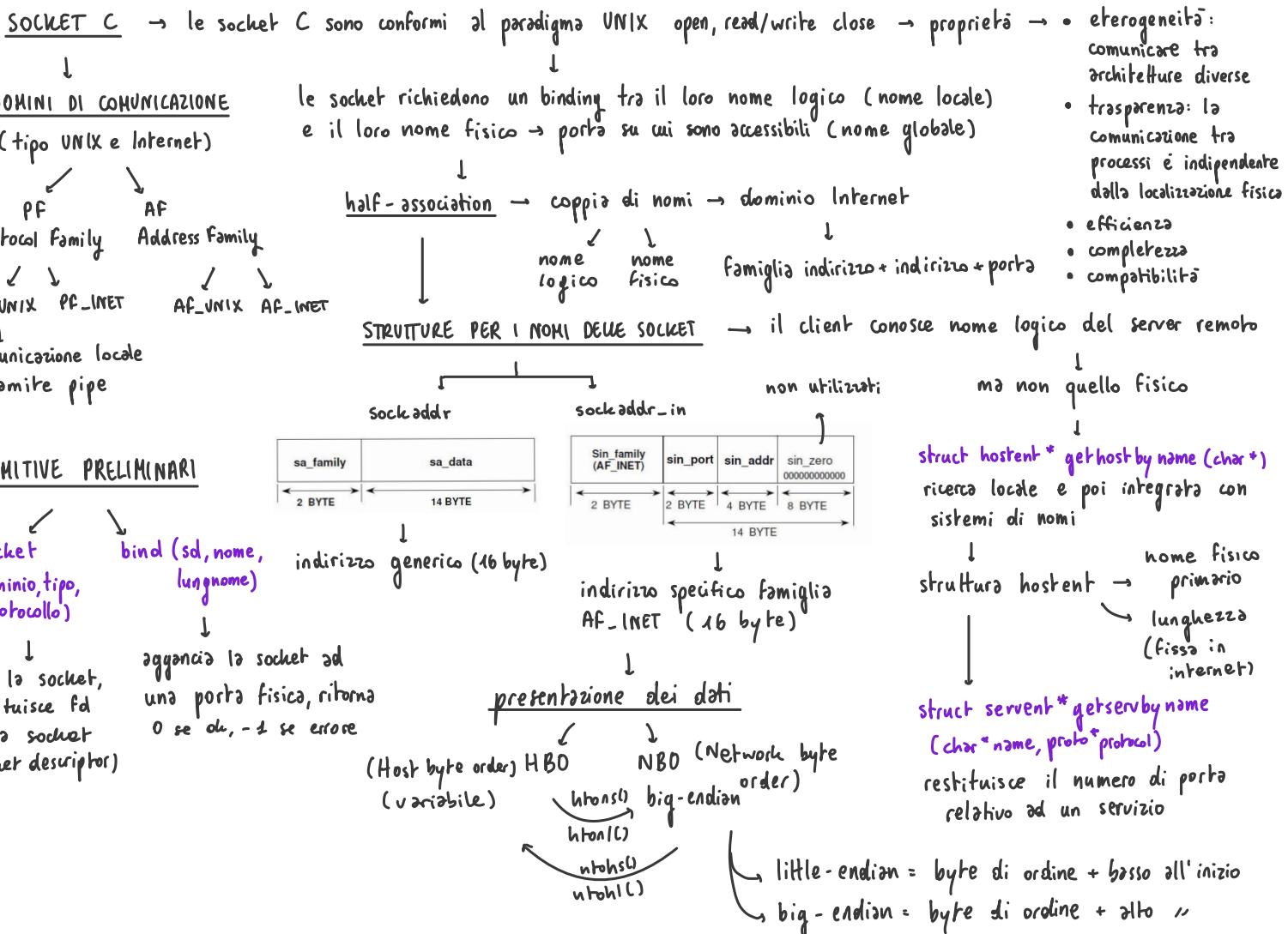
`SetSendBufferSize(int)`

↓

`SetReceiveBufferSize(int)`

si può inviare direttamente un pacchetto senza bufferizzato

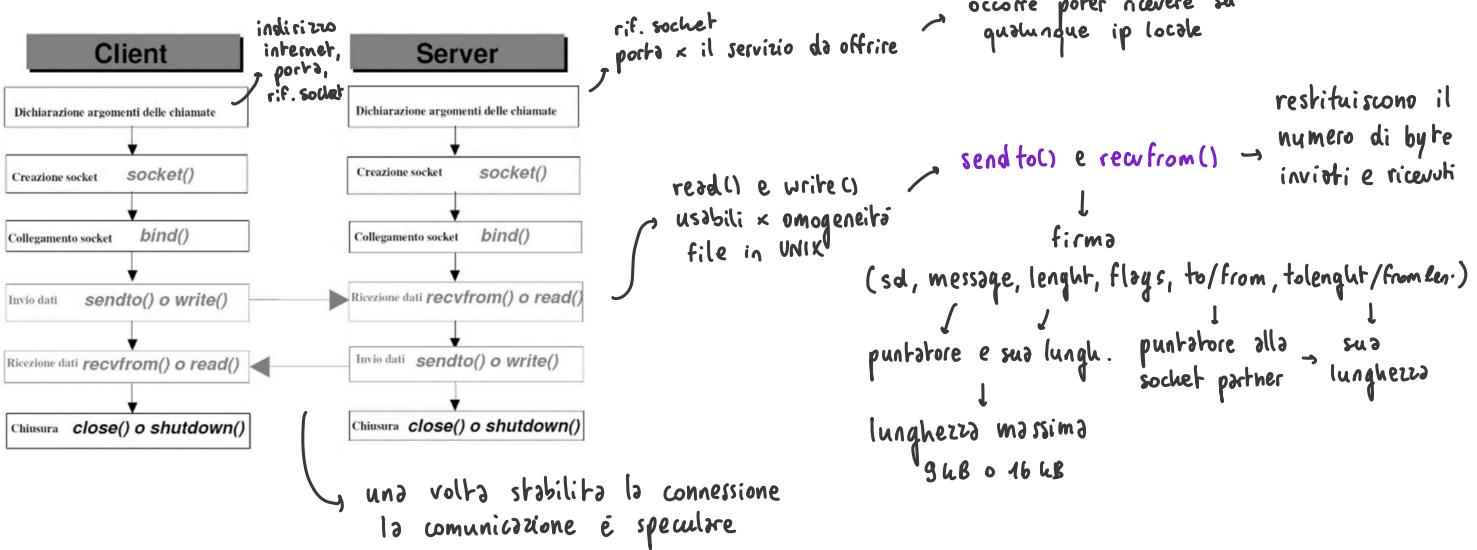
→ `ShutdownOutput()`



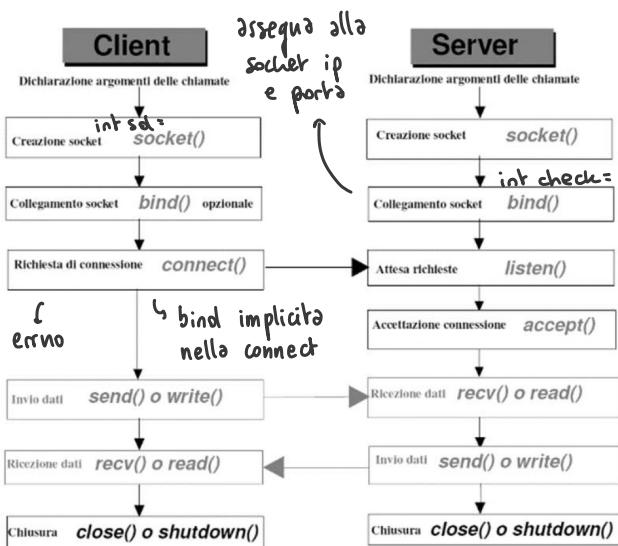
SOCKET DATAGRAM

↓

end-point di comunicazione che permettono di effettuare half-association relative ad un solo processo ma usabili x comunicare con chiunque nel dominio



SOCKET STREAM → prevedono una risorsa che rappresenta la connessione virtuale tra le entità interagenti



↓ CHIUSURA DELLE SOCKET

↓
la chiusura manda EOF all'altro, ogni end-point deve leggere fino alla fine del flusso

↓
primitiva **close()**

↓
locale, passante, istantanea

↓
decrementa il socket descriptor
così che il chiamante non può più utilizzarlo

↓
consegnata! (tcp)

buffer in uscita spedito, in entrata buttato (se non ancora ricevuto)

↓
dopo viene deallocated la memoria nel buffer

OZIONI PER LE PRIMITIVE → **getsockopt()** e **setsockopt()**

- **SO_SNDTIMEO** e **SO_RCVTIMEO**: permettono di cambiare la durata massima di una primitiva di send o receive dopo cui il processo viene sbloccato;
- **SO_SNDBUF** e **SO_RCVBUF**: permettono di cambiare la dimensione del buffer di trasmissione o ricezione, eventualmente eliminando attese per i messaggi di dimensioni elevate (la dimensione massima è di 64 kB);
- **SO_KEEPALIVE**: permette di controllare periodicamente la connessione: il protocollo di trasporto invia messaggi di controllo periodici per analizzarne lo stato;
- **SO_REUSEADDR**: permette di modificare il comportamento della bind(). Il sistema tende a non ammettere più di un utilizzatore alla volta di una porta, infatti ogni ulteriore utilizzatore viene bloccato; con questa opzione, si richiede che la socket sia senza controllo dell'unicità di associazione. In particolare, è utile quando si deve riavviare un server in seguito a un crash e deve essere operativo immediatamente.
- **SO_LINGER**: permette di modificare il comportamento della primitiva close(). A default, il sistema tende a mantenere dopo la close() la memoria in uscita anche per un lungo intervallo di tempo; con questa opzione è possibile modificare l'intervallo.

↓
prima fase asimmetrica (accettazione del client)
↓
poi fase simmetrica in cui C/S comunicano
↓
la connessione permane fino alla chiusura di una delle due half-association

↓
entità attiva entità passiva
↓
client che richiede servizio server che accetta il servizio e risponde

↓
primitiva di comunicazione sincrona
↓
connect()

✓ X
↓
errno

↓
ECONNREFUSED
ECONNABORTED
ETIMEDOUT
↓
errore di comunicazione nell'invio

deposita la richiesta nella coda del server
↓
connessione creata
↓
la coda deve essere creata!
↓
primitiva **listen()**

↓
locale, istantanea e senza attesa → fallisce solo se attuata su socket non valida
↓
prende come argomenti sd e lunghezza coda

↓
restituiscono il numero di byte effettivamente ricevuti o inviati (sd, message, length, flags)

↓
i dati non sono inviati con ogni primitiva ma bufferizzati da TCP (raggruppati e inviati alla prima comunicazione decisa dal driver TCP)

↓
x ovviare: mandare messaggi di lunghezza pari al buffer oppure opzione watermark

↓
ogni recv() restituisce i dati del driver locale e TCP no marcatori fine messaggio
↓
mandare max lunghezza fissa!

→ per RENDERE LE SOCKET ASINCRONE

↓
primitive **ioctl()** e **fcntl()**

↓
permettono azioni senza attesa e ad ogni cambiamento di stato nella socket (arrivo di dati) viene mandato il segnale SIGIO

↓
x rendere non bloccante: ioctl() con parametro FIONBIO a 1

di default
bloccante è >0

~ synchronous I/O multiplexing

SELECT

→ la select è una primitiva che permette di gestire l'attesa multipla sincrona → time-out intrinseco

↓
di base le
socket sono
sincrone bloccanti

le azioni di comunicazione su socket sono potenzialmente sospensive → la select sospende il processo

fino al primo evento o time-out

la select attende contemporaneamente più eventi
legati a più socket (o file) su cui le azioni
non siano bloccanti

scrittura, lettura
o eccezioni

↓
timeout = NULL
↳ 31 giorni

`int select (int nfds, fd-set * readfds, fd-set * writefds, fd-set * exceptfds, timeval * timeout)`

↓
all'invocazione segnala nelle
maschere gli eventi di
interesse ed il tempo

↓
al completamento ritorna
il numero di eventi occorsi
e indica quali con la
maschera di ingresso/uscita

↓
`int nfds` è il fd con
il numero più alto +1
nei tre set sorvegliati

↓
MASCHERE

↓
array di bit passati per riferimento

↓
maschere di read e except
sono sospensive perché è
necessario evento fatto da client

↓
quella di write può esserlo
nel caso il buf di ricezione
del client sia pieno

↓
eventi di
lettura

↓
rendono possibile e non
bloccante un'operazione

- dati da leggere con la `recv()`
- richiesta da accettare con `connect()`
- EOF o errore

↓
eventi di
scrittura

↓
operazione
completata

↓
anomalia
o eccezioni

↓
dati out-of-bounds
oppure close()

- connessione completa con la `connect()`
- quando si possono spedire altri dati con la `send()`
- se in una socket connessa il pari ha chiuso

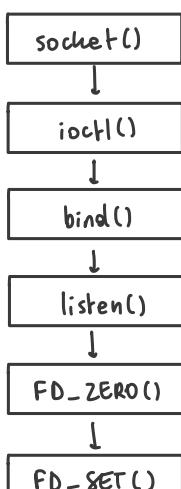
8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0
0	1	0	1	0	1	1	0	0
0	0	1	0	0	0	0	0	0

`nfds`
`readfds`
`writefds`
`exceptfds`

- qui `accept()` occupa il primo fd libero o ne aggiunge 1
- ogni `close()` libera un fd

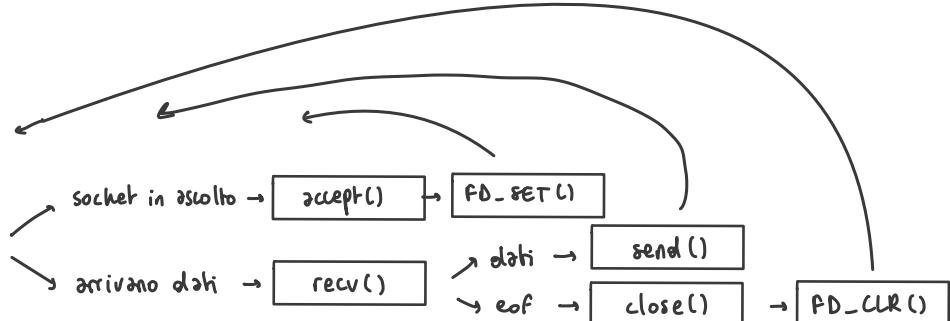
↓
la select lavora in symbiosi con
4 macro, che preparano l'ambiente
di esecuzione e testano i risultati

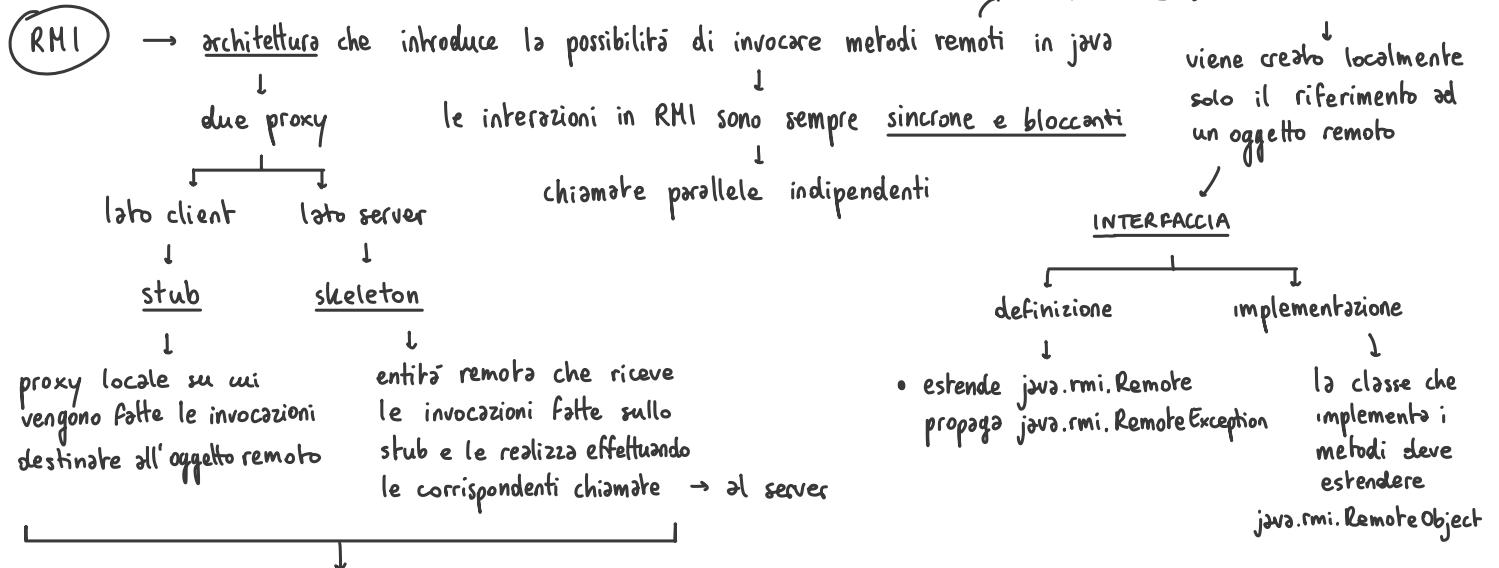
- `FD_SET (int fd, fd-set * set)` → setta a 1 l'fd nel set (aggiunge)
- `FD_CLR (int fd, fd-set * set)` → setta a 0 l'fd nel set (rimuove)
- `FD_ISSET (int fd, fd-set * set)` → cerca l'fd nel set e ritorna 1 se lo trova > 0 altrimenti
- `FD_ZERO (fd-set * set)` → inizializza tutto il set a 0



rende la
socket asincrona

↓
close()





Sono generati dal compilatore RMI e rendono possibile l'invocazione di un servizio remoto come se fosse locale

1. il client ottiene un rif remoto, attiva la chiamata dei metodi con lo stub e attende risultato
2. stub effettua serializzazione delle informazioni x la chiamata e le invia allo skeleton tramite astrazioni di RRL
3. skeleton effettua deserializzazione, invoca la chiamata sul server, serializza il valore di ritorno e lo invia allo stub
4. stub deserializza il valore di ritorno e lo restituisce al client

Remote Reference Layer

gestisce i rif agli ogg. remoti

oggetto remoto

entità a parte
si può creare il proprio
con `LocalRegistry`

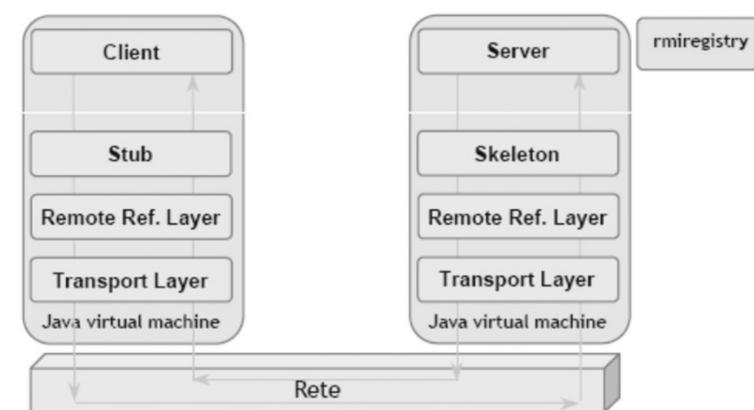
RMI Registry

→ ha la sua JVM

servizio di nomi che consente al server di pubblicare un servizio e al client di recuperare il proxy

mantiene al suo interno un insieme di coppie {name, ref.}

stringa che indica un servizio
indica su quale server si trova il servizio



livello di trasporto responsabile della gestione delle connessioni

↳ protocolli connection-oriented

gestito dalla JVM

→ CONCORRENZA → un thread x ogni richiesta

bytecode

→ COMUNICAZIONE → buon utilizzo delle risorse!

PASSAGGIO DEI PARAMETRI

trasforma oggetti complessi
`writeObject()` → OUTPUT

ACCESSO

client e server a tempo di compilazione ed esec. devono poter accedere alle risorse

se c'è già una connessione tra 2 JVM la si riutilizza

serializzazione → in sequenze di byte

`readObject()` → INPUT

client
interfaces a compilazione, stub esecuzione

connessioni sia sequenziali che multiplexing del canale

deserializzazione → decodifica la sequenza di

byte e ricostruisce una copia dell'oggetto originale

classe Naming

server
a comp. interface e implementazione, ad esecuzione stub e skeleton

byte e in uscita

se primitivo `readObject()` → INPUT

passaggio per valore → o Serializable

`readObject()` → INPUT

riferimento → se oggetto Remote

`readObject()` → INPUT

CLASS LOADER → carica le classi sia locali che remote

codebase = raccolta di file

sorgenti e risorse che comp. un oggetto

RMI ClassLoader → SICUREZZA

bind, rebind, ecc. non sono invoc. da remoto

ogni classloader può avere un SecurityManager

consulta file di policy e controlla le autorizzazioni di chi fa le chiamate

client e server processi distinti, ≠ spazio indirizzamento

RPC

- mettono a disposizione, tramite server, procedure che possono essere invocate in remoto dai vari client
 - ↳ viaggiano attraverso la rete → approccio applicativo di alto livello (+ OSI)

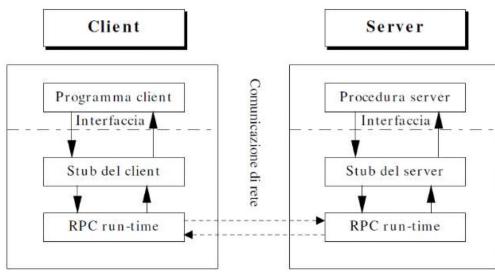
SEMANTICHE

1. at-least-once
(ritrasmessione a intervalli)
2. at-most-once (TCP)
3. exactly-once → reliability
4. may-be (best effort)
(UDP)

in RPC

1. si usa un time-out
2. si usano tabelle × azioni effettuate
3. l'azione viene fatta fino alla fine
4. time-out × il client

ARCHITETTURA



gli stub vengono generati automaticamente, servono due chiamate per ogni RPC

modello asimmetrico

- il client invoca lo stub, che si incarica di recuperare il server, marshallig dei parametri, richiesta al supporto run-time e trasporto
- il server riceve richiesta dallo stub, fa unmarshalling dei parametri e rimanda il risultato

marshalling livello 6 OSI (presentazione)

passaggio dei parametri sia × valore che × riferimento (oggetto rimane client e viene reso remoto)

main nello stub

→ del server

IDL - INTERFACE DEFINITION LANGUAGE

linguaggi per la descrizione di operazioni remote e la generazione degli stub

1. identificazione unica del servizio, usando un nome astratto e versioning
2. definizione dei dati × I/O con linguaggio astratto

SCOPO DEL IDL → supporto allo sviluppo e alla generazione automatica

RPC GEN

il client invia la richiesta ed attende in modo sincrono → il server può essere

sequenziale

parallelo

RPC ASINCRONE

(non bloccanti)

2 tipi

bassa latenza throughput elevato
↓
mandano un messaggio di richiesta e trascurano il risultato
differiscono l'invio delle richieste e le raggruppano in un unico msg

FAULT TOLERANCE

× mascherare i malfunzionamenti come perdita di mess. o crash

3 politiche

- aspettare per sempre
- timeout e ritrasmmissione (id)
- timeout e exception

MODALITÀ ASINCRONA IN SUN

→ per implementata serve:

- far usare TCP al client
- time-out nullo nel client
- server non deve prevedere risposte

clnt-call() e clnt-control()
timeout nullo, e nella risposta dichiarare <clnt_void> con 0

IMPLEMENTAZIONE RPC DI SUN

tre livelli di uso

alto

(livello utente client)
rusers()
get rpc port()

intermedio

per definire ed utilizzare nuovi servizi RPC × procedure singole

basso

gestione avanzata delle chiamate remote

callrpc() → client
registerpc() → server

associa id unico a procedura rem.

server → timeout
svc_register() → client
svc-run() → client
clntudp-create() → client
clnttcp-create() → client
clnt-call() → client
clnt-peer() → client
clnt-destroy() → client
↳ no divide socket

fà marshalling dei dati

XDR (IDL di SUN)

→ (6 OSI, presentazione)

gestisce tipi atomici e strutture dati

definizione di tipi di dati e delle specifiche di protocollo RPC

(numero di programma e versione)

id versione
id procedura
id programma
external number

int × tracciare il max

↳ 32 bit hex → app. comuni
2-3 → debug
4-5 → gen. dim

il resto: estensioni future

RPC BINDING → il binding prevede come ottenere l'aggancio corretto tra client e server → BINDER → NS

↓
2 modalità

- pessimistica statica → binding statico, a costo limitato ma poco flessibile
- ottimistica dinamica → costi maggiori, però load-balancing e crush avoidance
↓
dopo un primo legame si tende a riutilizzare lo stesso binding ottenuto come forse statico

↓
2 fasi

- ① fase statica di servizio → prima dell'esecuzione client specifica a chi vuole essere connesso (naming), poi binding
- ② fase dinamica di indirizzamento → a run-time si cercano eventuali server pronti x il servizio richiesto all'uso

↓
consente agganci flessibili e fornisce operazioni di:

- LOOKUP (servizio, vers, & servitore)
- REGISTER (servizio, vers, servitore)
- UNREGISTER (//)

↓
il binding è attuato come servizio coordinato di più server

↓
la RISOLUZIONE DEI NOMI può essere

↓
esplicita

↓
client fa richiesta broadcast x trovare server e accetta 2^a risposta

↓
implicita

↓
"binder"
si usa name server con tabelle di binding

SISTEMI DI NOMI

↓
proprietà

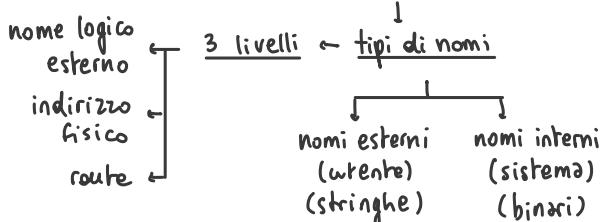
1. generalità (ampia varietà di nomi)

2. definizioni multiple (+ nomi diversi x 1 oggetto)

3. distribuibilità → direttori possono essere partitionati/replicati

4. user-friendliness

↓
nomi di alto livello (facile comprensione)



SPAZI DEI NOMI

↓
piatto partitionato descrittivo

no struttura, x pochi utenti e entità

gerarchie e contesti
DNS

struttura di oggetti caratterizzati da attributi per id. l'entità corrispondente

COMPONENTI DI UN SISTEMA DI NOMI

↓
client name server

↓
chi deve riservare un nome e le risorse che devono essere rese note

↓
fornisce la tabella di corrispondenza

tuple di attributi

ARCHITETTURA

gestore partizionato

gestore replicato

gestore centralizzato

riferimenti divisi tra i gestori stesso rif. su + gestori

+ di 1 NS

problem di consistenza

necessario COORDINAMENTO:

- distribuzione dei nomi
- risoluzione dei nomi
- politiche di caching
- politiche di routing
- propagazione di conoscenza tra vicini

richieste da client a server due slave:

1. trovare autorità corretta
2. verificare autorizzazioni
3. eseguire l'operazione

DIRECTORY X.500

servizio standard di directory e di nomi organizzato in albero logico → DIT (Div. information tree)

ogni elemento → DN → // base → insieme di tutte le informazioni del sistema

↓
com it edu
cineca unibo
 oisi

↓
id univoco parte relativa nel percorso DIT

↓
RICERCATE sul DIT tramite query SQL

op. raccomandate lettura, scrittura non efficienti

DUA : Directory User Agent

DSA : Directory System Agent

DAP : Directory Application Protocol

DSP : Directory System Protocol

