Interprocess Communication

Sommario

- Processi (e *thread*) cooperanti
- Il paradigma produttore-consumatore
- *Shared Memory* e *Inter Process Communication* (IPC) *facility*
- Proprietà caratteristiche della comunicazione
- Comunicazione diretta e comunicazione indiretta
- Mailbox: condivisione, proprietà
- Comunicazione: sincronizzazione e bufferizzazione
- Comunicazione client-server: *Socket*, *Remote Procedure Call* (RPC), *Remote Method Invocation* (RMI) in Java
- *Marshalling* e *stub*

Processi (e thread) Cooperanti

Processi indipendenti non possono modificare o essere modificati dall'esecuzione di un altro processo.

Processi cooperanti possono modificare o essere modificati dall'esecuzione di altri processi.

Vantaggi della cooperazione tra processi:

- **Condivisione delle informazioni** ► più utenti possono essere interessati alle stesse porzioni di informazioni
- **Aumento della velocità di computazione** (parallelismo)

  ► Suddivisione di un job in sotto-attività per una esecuzione in parallelo (in caso di multiplazione di una o più risorse)
- **Modularità**
- **Praticità di realizzazione/utilizzo** ► Un singolo utente può lavorare su più attività contemporaneamente

Commmunicating cooperating processes

Il paradigma **produttore-consumatore**

- La concorrenza implica la presenza di memorie tampone (***buffer***) riempite dal produttore e vuotate dal consumatore. Il buffer di comunicazione può essere fornito dal SO tramite la **Inter Process Communication (IPC) facility** o può essere esplicitamente codificato dal programmatore facendo uso di una **Shared Memory**.
- Il parallelismo è determinato dalla possibilità che il produttore produca un elemento mentre il consumatore ne sta consumando uno prodotto in precedenza.
  - *Buffer illimitato (unbounded-buffer)*: non c'è un limite teorico alla dimensione del buffer. Il solo limite è costituito dal fatto che il consumer può dover attendere che il producer introduca nuovi elementi nel buffer
  - *Buffer limitato (bounded-buffer)*: la dimensione del buffer è fissata. Il consumer deve attendere se il buffer è vuoto, mentre il producer deve attendere se il buffer è pieno
- Il paradigma producer-consumer implica l'adozione di meccanismi di sincronizzazione tra i processi coinvolti.

Cooperating Processes: Shared-Memory

Shared data

```
#define BUFFER_SIZE 10
typedef struct { . . . } item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Producer process

```
item nextProduced;
    while (1) {
                while (counter == BUFFER_SIZE); /* do nothing */
                buffer[in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                counter++;
               }
```

Consumer process

```
Item nextConsumed;
while (1) {
             while (counter == 0); /* do nothing */
             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             counter--;
           }
```

Cooperating Processes: Inter Process Communication (IPC)

IPC è una *facility* di comunicazione e interazione tra processi (e thread)

- Come può un processo passare informazioni ad un altro?
  - Message-Passing → un processo comunica con un altro senza ricorrere a dati condivisi (non si condivide lo spazio di indirizzi)
  - Se i processi P e Q vogliono comunicare, devono stabilire un canale di comunicazione (fisica o logica) tra di loro e usare le due operazioni:
    - ✓ **`send`** *`(messaggio)`* → la dimensione del messaggio può essere fissa o variabile.
    - ✓ **`receive`** *`(messaggio)`*
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

Cooperating Processes: Inter Process Communication (IPC)

Problematiche dello scambio di messaggi

- **Affidabilità**: i canali possono essere inaffidabili (es: reti). Bisogna implementare appositi protocolli *fault-tolerant* (basati su *acknowledgment* e *timestamping*).
- **Autenticazione**: come autenticare i due partner?
- **Sicurezza**: i canali utilizzati possono essere intercettati.
- **Efficienza**: se prende luogo sulla stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e dei semafori.

Cooperating Processes: Inter Process Communication (IPC)

- Dal punto di vista logico, un link di comunicazione può essere realizzato in diversi modi:
  - Comunicazione Diretta (tramite i nomi dei processi) o Indiretta (tramite *mailbox* o *porta*)
  - Comunicazione Simmetrica o Asimmetrica (il ricevente conosce il mittente o no)
  - Comunicazione Sincrona o Asincrona (primitive bloccanti o non bloccanti)
  - Comunicazione Estesa o a rendez-vous limitato (se sia la send che la receive sono bloccanti o no)
  - Comunicazione Con coda Non bufferizzata o Bufferizzata (con capacità limitata o no).

Direct communication

- I processi devono conoscere esplicitamente il nome del destinatario o del mittente della comunicazione:

Indirizzamento simmetrico

- **send** (*P, message*) – manda un messaggio al processo P
- **receive**(*Q, message*) – riceve un messaggio dal processo Q

Indirizzamento asimmetrico

- **send** (*P, message*) – manda un messaggio al processo P
- **receive**(*id, message*) – riceve un messaggio da un processo qualunque (la variabile ***id*** conterrà di volta in volta il nome del processo con cui si comunica)

- Proprietà di un canale di comunicazione:
  Le connessioni sono stabilite automaticamente.
  Una connessione è associata esattamente a due processi.
  Fra ogni coppia di processi esiste esattamente una connessione.
  La connessione può essere unidirezionale, ma di norma è bidirezionale.

Indirect Communication

I messaggi sono mandati e ricevuti attraverso **mailbox** (o porte).

- Ciascuna mailbox ha un identificatore univoco.
- I processi possono comunicare solo se hanno una mailbox condivisa.

Il canale di comunicazione ha le seguenti proprietà:

- viene stabilita una connessione fra due processi solo se entrambi hanno una mailbox condivisa.
- una connessione può essere associata a più di due processi.
- fra ogni coppia di processi comunicanti possono esserci più connessioni. Ciascuna di esse corrisponde ad una mailbox.
- la connessione può essere unidirezionale o bidirezionale.

Operazioni:

- creare una mailbox;
- mandare e ricevere messaggi per mezzo della mailbox;
- cancellare una mailbox.

Le primitive sono definite come:

- **`send`** `(A, messaggio)`: manda un messaggio alla mailbox *A*;
- **`receive`** `(A, messaggio)`: riceve un messaggio dalla mailbox *A*.

Condivisione di una mailbox

La ***send*** si blocca se la mailbox è piena; la ***receive*** si blocca se la mailbox è vuota.

S'immagini che P_1, P_2, e P_3 condividano una mailbox *A*.

- P_1 invia un messaggio ad *A*; P_2 e P_3 eseguono una *receive* da *A*.
- Quale processo riceverà il messaggio spedito da P_1?
- La risposta dipende dallo schema che si sceglie:
  - permettere che una connessione sia associata con al più due processi.
  - permettere ad un solo processo alla volta di eseguire un'operazione di receive.
  - permettere al sistema di decidere arbitrariamente quale processo riceverà il messaggio. Il sistema può anche notificare il ricevente al mittente.

Proprietà di una mailbox

Una mailbox può appartenere:

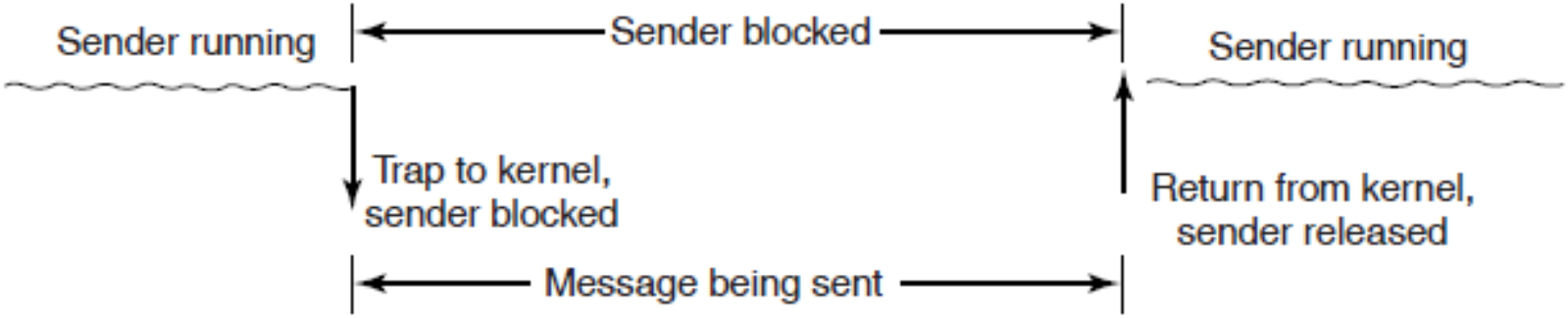
- al sistema operativo
  - ha una esistenza indipendente non correlata ad alcun processo
- ad un processo
  - è parte dello spazio di indirizzi di un processo
  - si distingue tra proprietario di una mailbox (può solo riceverne i messaggi) e utente (può solo inviarvi messaggi)
  - una mailbox ha come UNICO proprietario il processo che la crea
    - non esiste possibilità di confusione tra i destinatari di un messaggio
  - alla terminazione di un processo viene meno la mailbox ad esso associata
    - i processi che inviano messaggi ad una mailbox non più esistente ricevono una notifica di errore dal sistema operativo.

Communication synchronization

Il passaggio di messaggi può essere **bloccante** . . . . . oppure

Invio/ricezione bloccanti: sono considerati **sincroni**.

- **Invio bloccante**: il processo che invia viene bloccato finché il messaggio viene ricevuto dal processo che riceve o dalla mailbox.
- **Ricezione bloccante**: il ricevente si blocca sino a quando un messaggio non è disponibile.



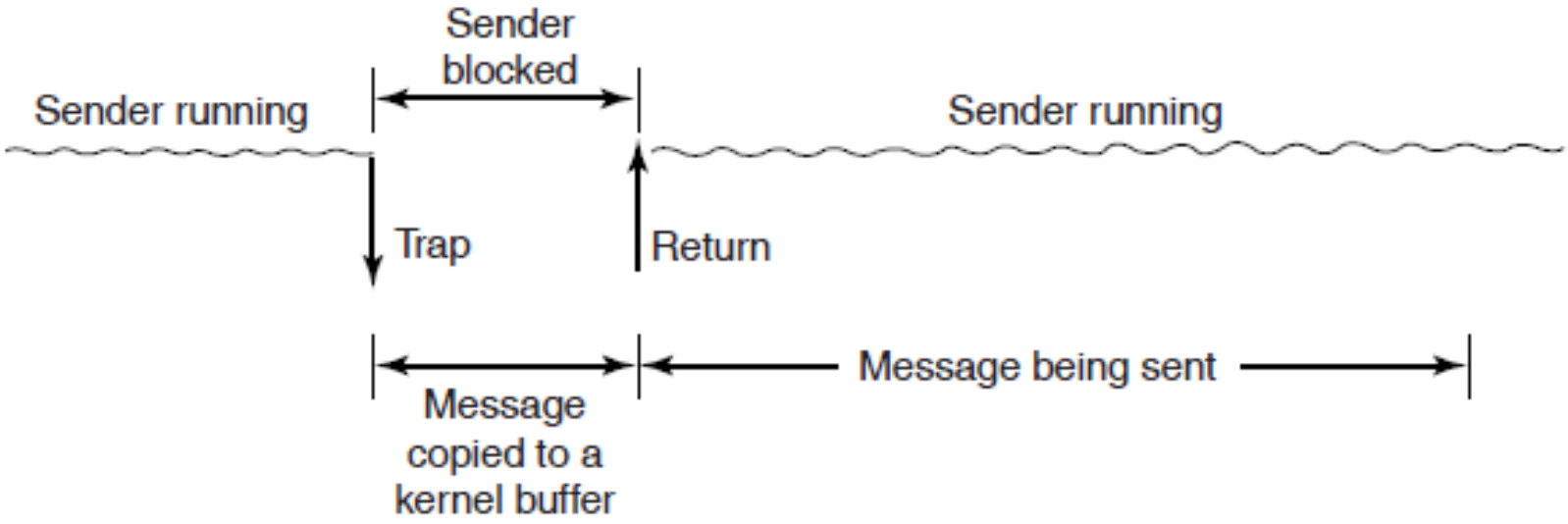
In caso di send() e receive() bloccanti si parla di **rendez vous esteso** fra mittente e destinatario.

Communication synchronization

. . . . . il passaggio di messaggi può essere **non bloccante**.

Invio/ricezione non bloccanti: sono considerati **asincroni**.

- **Invio non bloccante**: il processo che invia manda il messaggio e riprende l'attività.
- **Ricezione non bloccante**: il ricevente acquisisce un messaggio o valido o nullo.



In caso di send() e receive() non bloccanti si parla di **rendez vous limitato** fra mittente e destinatario.

Communication buffering

- I messaggi scambiati in fase di comunicazione risiedono in una coda temporanea.

Esistono tre possibilità per implementare il buffer dei messaggi relativo ad un link:

1) Capacità zero – lunghezza massima della coda: 0 messaggi.
   - La connessione non potrà avere nessun messaggio in attesa nella coda.
   - Il mittente deve bloccarsi finché il destinatario riceve il messaggio (rendez vous).
2) Capacità limitata – coda a lunghezza finita di *n* messaggi.
   - Se la coda non è piena, un nuovo messaggio viene inserito nella coda. Il messaggio viene copiato o ne viene mantenuto un puntatore.
   - Il mittente deve bloccarsi solo se la coda di comunicazione è piena, altrimenti può continuare l'esecuzione senza attendere.
3) Capacità illimitata – coda di lunghezza potenzialmente illimitata.
   - Un qualunque numero di messaggi può essere accodato.
   - Il mittente non si blocca mai.

Client-Server Communication

La comunicazione nei sistemi client-server può essere implementata secondo gli schemi canonici oppure mediante ulteriori strategie:

- ***Socket***
- **Chiamata di procedura remota --- *Remote Procedure Call* (RPC)**
- **Invocazione di metodo remoto --- *Remote Method Invocation*** (RMI Java)

Client-Server Communication: il Socket

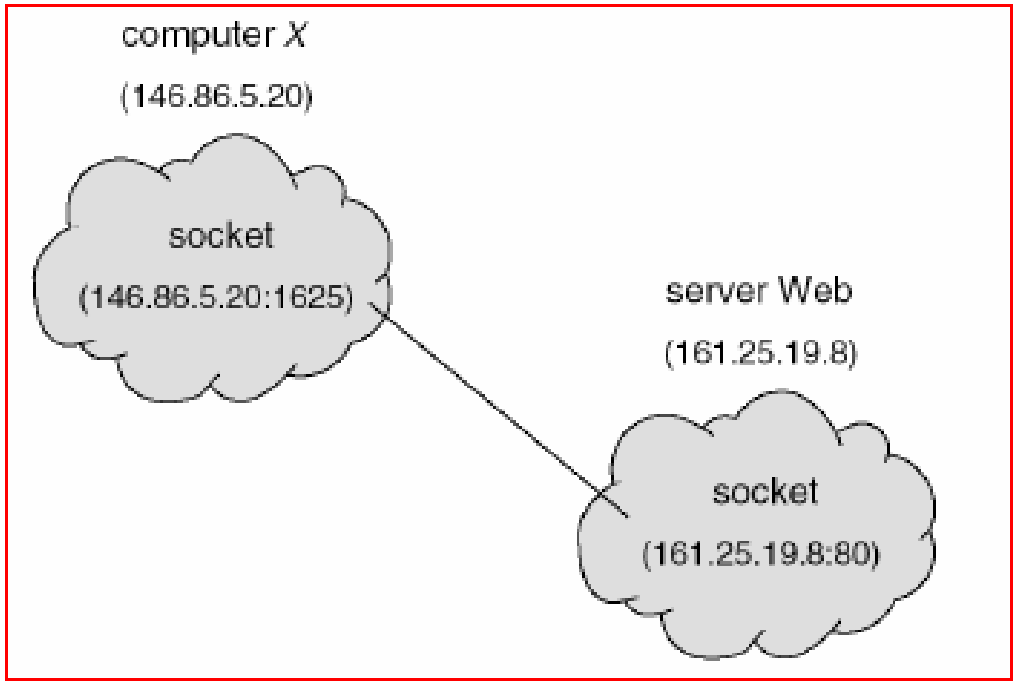
Un **socket** è definito come un estremo (***endpoint***) di un canale di comunicazione ed è associato a un indirizzo IP: esso è identificato da un indirizzo IP concatenato con un numero di porta.

Il socket **146.86.5.2:1625** fa riferimento alla porta **1625** on host **146.86.5.2**

Una comunicazione consiste in una coppia di socket.

Un server resta in ascolto su di una data porta per l'arrivo di richieste da un client, accetta la connessione richiesta dal client e completa la connessione.

I server che implementano servizi specifici (telnet, ftp, http) ascoltano sulle cosiddette *well-known ports* (port < 1024).



Client-Server Communication nei multicomputer

La Remote Procedure Call (RPC)

La chiamata di procedura remota (RPC) è **il modello di computazione distribuita più astratto**, che estende il meccanismo di chiamata di procedura al caso di sistemi connessi in rete.

Idea di base: un processo su una macchina può richiedere l'esecuzione di codice su una CPU remota. L'esecuzione di procedure remote deve apparire simile a quella di procedure locali.

- La RPC è, infatti, una IPC basata su scambio di messaggi ben strutturati. Ciascun messaggio è indirizzato verso un **demone RPC** che ascolta su una porta del sistema remoto. Esso contiene:
  - un identificativo della funzione da eseguire;
  - i parametri da passare.
- L'output della procedura viene restituito al client.

Nasconde (in parte) all'utente la delocalizzazione del calcolo: l'utente non esegue mai send/receive, ma deve solo scrivere e invocare procedure come al solito.

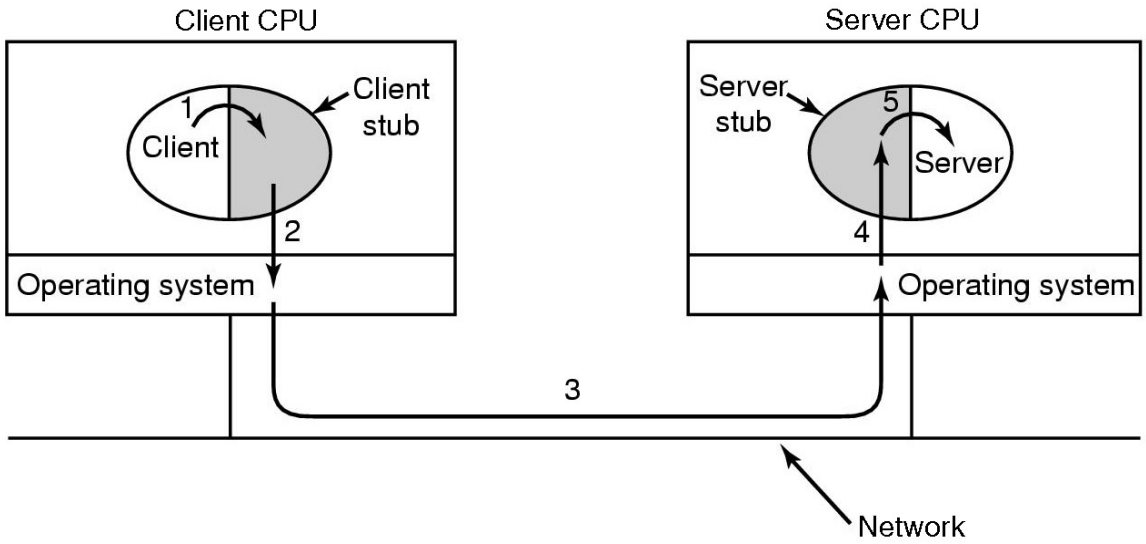
Client-Server Communication: La Remote Procedure Call (RPC)

Stub – terminale remoto sul lato client della procedura lato server.

- Ogni procedura remota mantiene uno stub client.
- Lo stub del lato client localizza il server, gestisce l'agreement della connessione attraverso la porta sul server e compie la traduzione (**marshalling**) dei parametri secondo un formato comprensibile dal destinatario.
- Lo stub dal lato server riceve il messaggio, estrae i parametri tradotti ed invoca la procedura sul server.

☞ criteri realizzativi

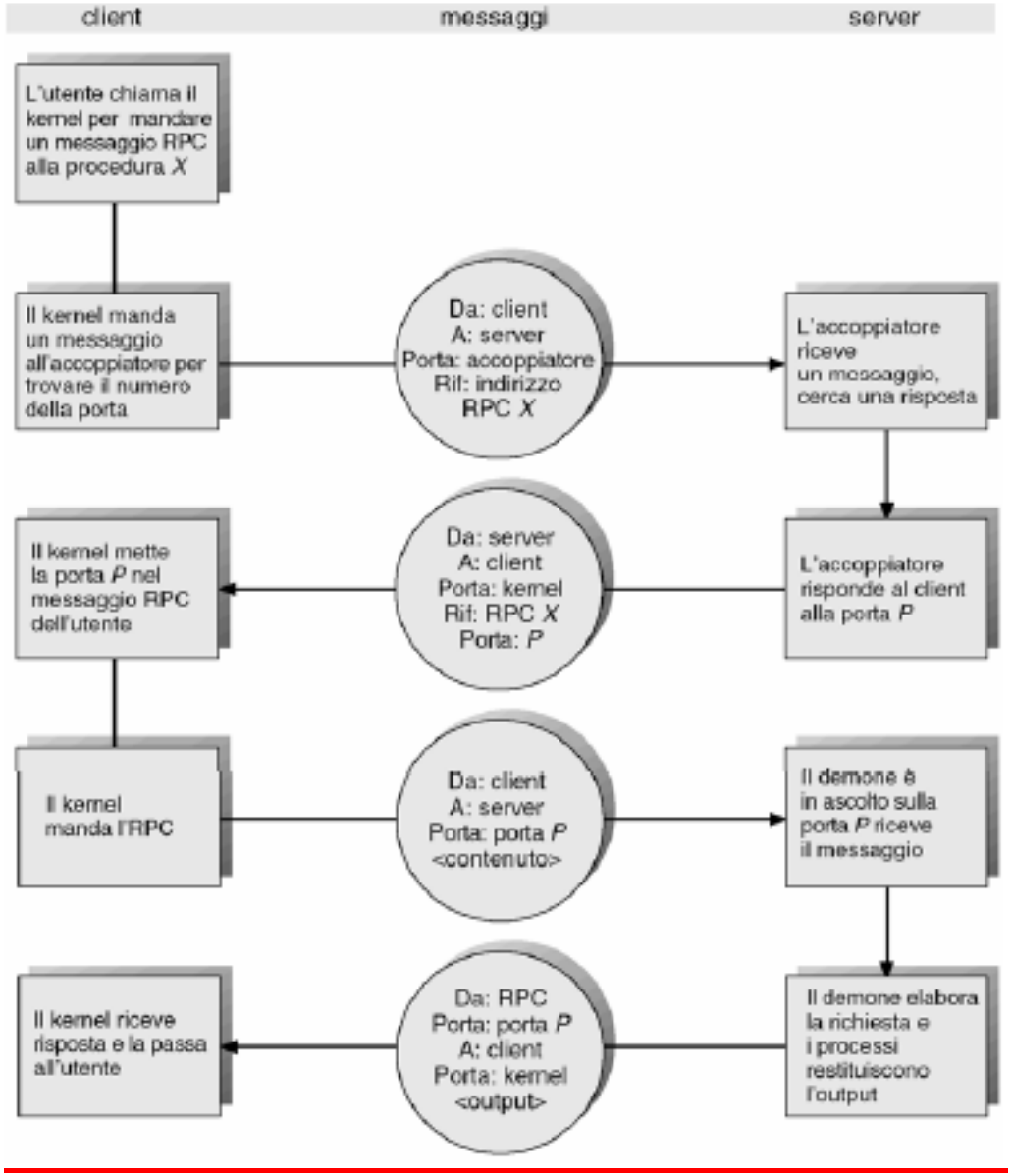
- non si possono passare puntatori
  - *call by reference* diviene *call by copy-restore*
- linguaggi debolmente tipizzati
- non è sempre possibile stabilire i tipi dei parametri
- non si possono usare variabili globali



→ marshalling

Client-Server Communication

Remote Procedure Call



Client-Server Communication: La Remote Method Invocation (RMI)

Invocazione di metodo remoto

- L'invocazione di metodo remoto è un meccanismo di Java simile alla RPC.
- RMI permette ad un programma Java di invocare metodi relativi a oggetti remoti.
- RMI vs. RPC
  - RPC supporta solo programmazione procedurale;
  - RMI rende possibile passare come parametri dei metodi remoti oltre che strutture dati ordinarie.
- La RMI implementa i metodi remoti usando:
  - **Stub** (componente lato client del metodo remoto, responsabile della creazione di un *codice* contenente nome del metodo da invocare e parametri su cui è eseguito il marshalling)
  - **Skeleton** (componente lato server responsabile della traduzione dei parametri e dell'invocazione del metodo desiderato)