UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

Alberto Ceselli
(alberto.ceselli@unimi.it)

Informatica II
Sistemi Operativi
DIGIP - a.a. 2015/16

Sistemi Operativi

(modulo di Informatica II)

La comunicazione tra processi

Patrizia Scandurra

Università degli Studi di Bergamo

a.a. 2014-15

Sommario

- Processi cooperanti
- La comunicazione tra processi
  - Necessità e caratteristiche
  - Implementazione
    - a *memoria condivisa*
    - *scambio di messaggi*
- La comunicazione tramite *scambio di messaggi*
- La comunicazione in *ambito distribuito*

Processi cooperanti (1)

- Un ***processo indipendente*** *non può influenzare o essere* influenzato dagli altri processi in esecuzione

- Un ***processo cooperante*** *può influenzare o essere influenzato da* altri processi in esecuzione nel sistema

Processi cooperanti

- Hanno uno scopo applicativo comune
- Possono condividere informazioni
- Possono influenzare o essere influenzati da altri processi

- Coordinamento della computazione → Sincronizzazione
- Scambio di informazioni → Comunicazione

Comunicazione

Inter-Process Communication (IPC)

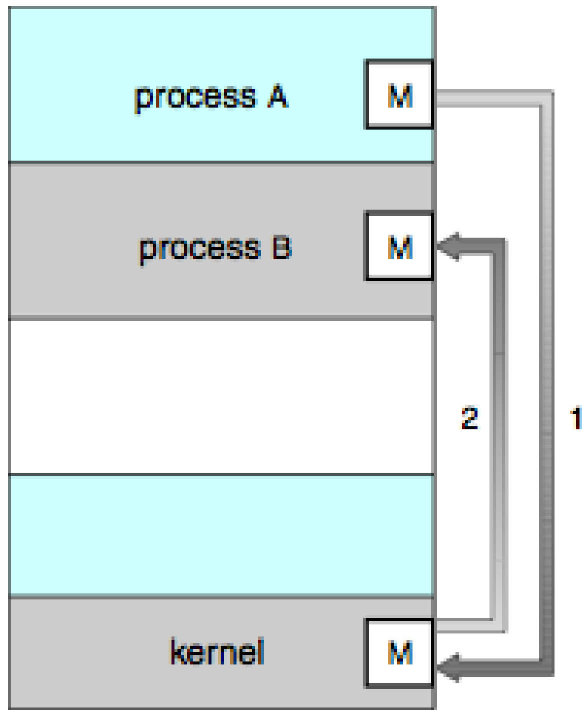
- Meccanismi e politiche
  - che permetto ai processi di scambiarsi informazioni
  - per elaborare in modo cooperativo
- Necessità
  - Trasferimento di informazioni da processo mittente a ricevente
  - Condivisione di informazioni

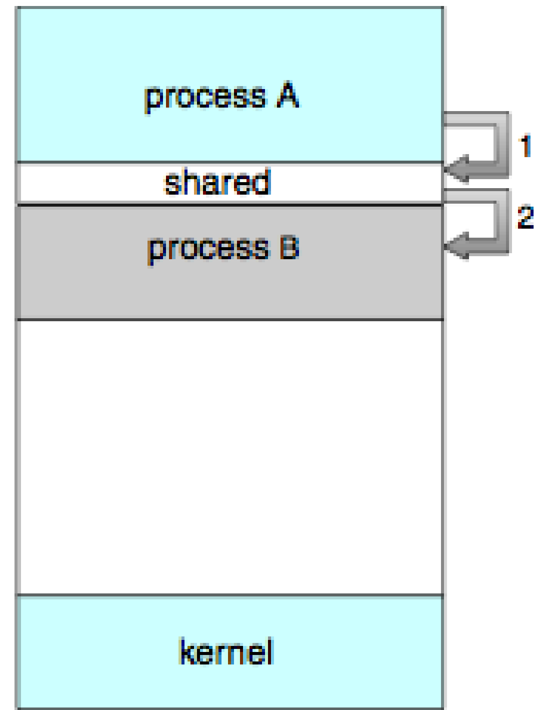
IPC: Caratteristiche

- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità
- Semplicità di uso nelle applicazioni
- Omogeneità delle comunicazioni
- Affidabilità
- Sicurezza
- Protezione
- Integrazione nel linguaggio di programmazione

IPC: modelli

Scambio di messaggi **Memoria condivisa**





Memoria condivisa

- **Caratteristiche**:
  - Condivisione di un segmento di memoria
  - Condivisione buffer di comunicazione (vedi producer-consumer)
- **Problemi**:
  - **Identificazione dei processi** comunicanti (comunicazione diretta)
  - **Consistenza** degli accessi
    - Lettura e scrittura sono incompatibili tra loro
  - Richiede **sincronizzazione dei processi** per accesso in *mutua esclusione*

Memoria condivisa in Unix (1)

```
1.  #include <pthread.h>
2.  #include <stdio.h>
3.  #include <unistd.h>
4.  #include <sys/types.h>
5.  #include <sys/shm.h>
6.  #include <sys/stat.h>
7.  #include <stdlib.h>

8.   int main()
9.   {
10.      pid_t pid;
11.      /* the identifier for the shared memory segment */
12.      int segment_id;
13.      /* a pointer to the shared memory segment */
14.      char* shared_memory;
15.      /* the size (in bytes) of the shared memory segment */
16.      const int segment_size = 4096;

17.       /** allocate  a shared memory segment, returns identifier */
18.      segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);
19.      /* fork a child process */
20.      pid = fork();
21.
22.      /** attach the shared memory segment */
23.      shared_memory = (char *) shmat(segment_id, NULL, 0);
24.
25.      printf("shared memory segment %d attached at address %p –
26.           Process %d \n", segment_id, shared_memory, pid);
```

`IPC_PRIVATE`
Indica l'identificatore del segmento di memoria: in questo caso un segmento condiviso
`S_IRUSR | S_IWUSR`
significa che il propietario del segmento può leggere e scrivere

`shmat` restituisce il puntatore alla prima cella della memoria.
Parametro 2: il SO decide dove annettere la memoria
Parametro 3: 0 sola lettura, >0 scrittura

Memoria condivisa in Unix (2)

```
27.     if (pid < 0) { /* error occurred */
28.         fprintf(stderr, "Fork Failed\n");
29.         exit(-1);
30.     }else if (pid == 0) { /* child process */
31.         /** now print out the string from shared memory */
32.         usleep(500);
33.         printf("I am the child -- *%s*\n", shared_memory);
34.         /** now detach the shared memory segment */
35.         if ( shmdt(shared_memory) == -1) {
36.                 fprintf(stderr, "Unable to detach\n");
37.         }
38.         exit(0);
39.     }else { /* parent process */
40.         /** write a message to the shared memory segment   */
41.         sprintf(shared_memory, "Hi there!");
42.         wait(NULL);
43.         if ( shmdt(shared_memory) == -1) {
44.                 fprintf(stderr, "Unable to detach\n");
45.         }
46.         /** now remove the shared memory segment */
47.         shmctl(segment_id, IPC_RMID, NULL);
48.         exit(0);
49.    }
50. }
```

Scambio dei messaggi

- La memoria condivisa richiede al programmatore di implementare il codice per la realizzazione e la gestione della memoria condivisa
- Nella comunicazione basata su scambio di messaggi queste problematiche sono gestite dal SO
  - È particolarmente utilizzato in contesti distribuiti
- **Caratteristiche**
  - Identificazione dei processi comunicanti
  - Memoria non condivisa tra processi
  - Instaurazione di un canale di comunicazione su cui spedire i messaggi
- **Problemi**
  - Sincronizzazione per l'accesso ai messaggi
  - gestita però implicitamente dal SO fornendo due operazioni:
    - **send** (messaggio)
    - **receive** (messaggio)

Scambio dei messaggi -Messaggi

- Contenuto
  - Processo mittente
  - Processo destinatario
  - Informazioni da trasmettere
  - Eventuali altre informazioni di gestione dello scambio messaggi

- Dimensione
  - Fissa
  - Variabile

Scambio dei messaggi - Canali

- Se due processi vogliono comunicare, devono:
  - stabilire un **canale di comunicazione** tra di loro
  - scambiare messaggi mediante **send/receive**

- Implementazione di un canale di comunicazione:
  - fisica (come memoria condivisa, hardware bus) o
  - logica (come le proprietà logiche)

Scambio dei messaggi – Implementazione (1)

- **Domande**
  - Come si stabiliscono le connessioni (canali)?
  - Una connessione può essere associata a più di due processi?
  - Quante connessioni possono esserci fra ogni coppia di processi?
  - Cos'è la capacità di una connessione?
  - La dimensione di un messaggio che una connessione può ospitare è fissa o variabile?
  - Una connessione è unidirezionale o bidirezionale?

Scambio dei messaggi – implementazione (2)

1. La denominazione dei processi
   - **comunicazione diretta**
   - **comunicazione indiretta**
2. La sincronizzazione
   - Il passaggio dei msg può essere **bloccante** oppure **non bloccante** (ovvero **sincrono** oppure **asincrono**)
3. La bufferizzazione
   - i messaggi scambiati risiedono in una coda temporanea

Scambio dei messaggi – la comunicazione diretta

- I processi devono conoscere esplicitamente il nome del destinatario o del mittente:
  - **send (P, msg)** – manda un messaggio al processo P
  - **receive (Q, msg)** – riceve un messaggio dal processo Q

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

Scambio dei messaggi – la comunicazione indiretta (1)

- **I messaggi sono mandati e ricevuti attraverso una mailbox o porte**
  - Ciascuna mailbox ha un identificatore univoco
  - I processi possono comunicare solo se hanno una mailbox condivisa
  - Le primitive sono definite come
    - **send** (A, msg) – manda un messaggio alla mailbox A
    - **receive** (A, msg) – riceve un messaggio dalla mailbox A

Scambio dei messaggi – la comunicazione indiretta (2)

- Proprietà di un canale di comunicazione :
  - Viene stabilita una connessione fra due processi solo se entrambi hanno una mailbox condivisa
  - Una connessione può essere associata a più di due processi (non binaria)
  - Fra ogni coppia di processi comunicanti possono esserci più connessioni
  - La connessione può essere unidirezionale o bidirezionale

Scambio dei messaggi – la comunicazione indiretta (3)

- Proprietario di una mailbox: un processo o il SO
  - Se il proprietario è un processo:
    - Risiede nel suo spazio di indirizzi
    - Quando il processo proprietario termina, la mailbox viene deallocata
    - Il **proprietario** può solo ricevere msg dalla mailbox
    - Gli altri processi (**utenti**) possono solo inviare msg alla mailbox
  - Se il proprietario è il SO, occorrono ulteriori primitive per:
    - **creare una mailbox**
      - inizialmente il processo che la crea è l'unico ricevente (proprietario di default)
    - **cancellare una mailbox**
    - **trasferire il privilegio di ricevere** msg dalla mailbox
      - per dar luogo a più processi riceventi per mailbox

Scambio dei messaggi – la comunicazione indiretta (4)

Esempio di scenario di condivisione di una mailbox:

- P1, P2, e P3 condividono una mailbox A
- P1, invia un messaggio ad A; P2 e P3 eseguono una receive da A

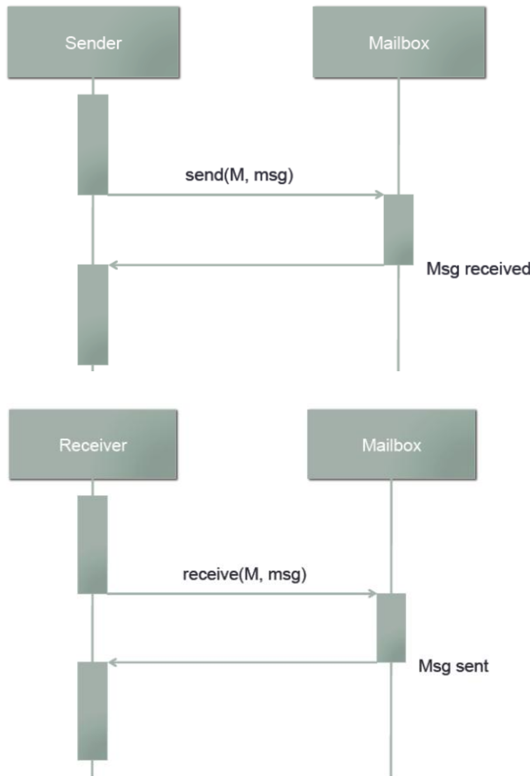
- Quale processo riceverà il messaggio spedito da P1?
- La risposta dipende dallo schema implementativo adottato:
  - Permettere che una connessione sia associata con al più due processi
  - Permettere ad un solo processo alla volta di eseguire un'operazione di receive
  - Permettere al sistema di decidere arbitrariamente quale processo riceverà il messaggio. Il sistema può anche notificare il ricevente al mittente

Scambio dei messaggi – la comunicazione indiretta (5)

Comunicazioni con molti possibili mittenti o riceventi

- Comunicazioni da molti mittenti a un ricevente
- Comunicazioni da un mittente a molti possibili riceventi
- Comunicazioni da molti mittenti a molti possibili riceventi

Scambio dei messaggi – la sincronizzazione

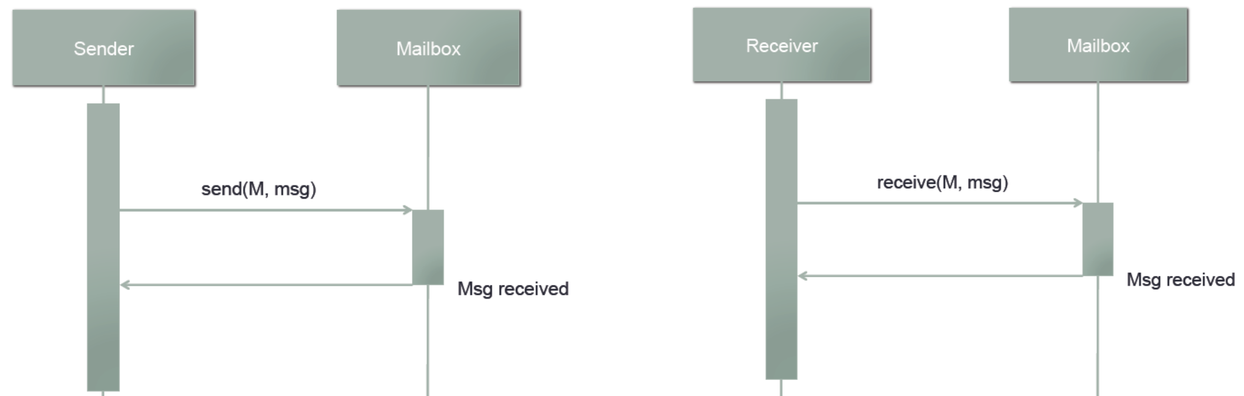


Lo scambio può essere bloccante (sincrono) oppure non bloccante (asincrono)

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

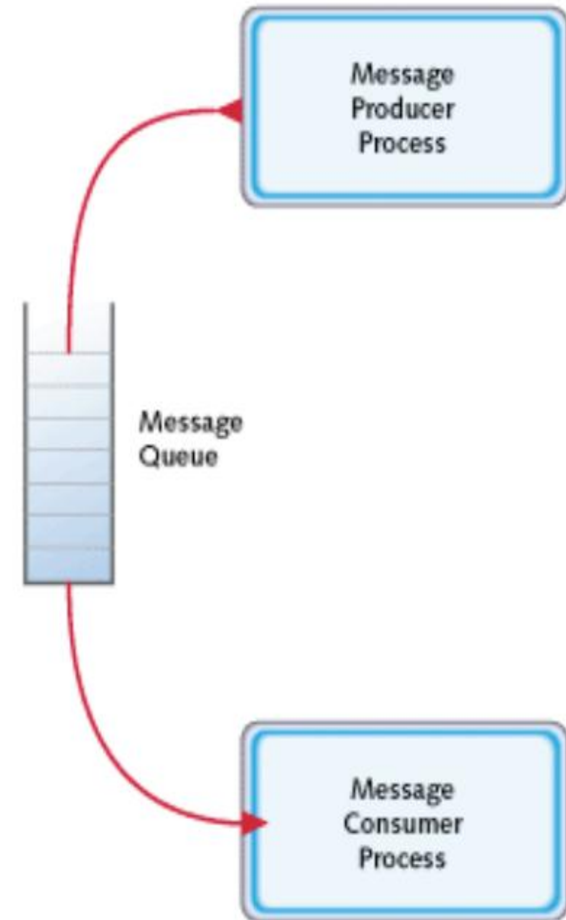
RENDEZVOUS

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



Scambio dei messaggi – la bufferizzazione (1)

- Tre modi per implementare la coda temporanea:
  1. **Capacità zero – 0 messaggi**
     Il mittente deve bloccarsi finchè il destinatario riceve il messaggio (rendezvous)
  2. **Capacità limitata** – lunghezza finita di n messaggi
     Il mittente deve bloccarsi se la coda è piena
  3. **Capacità illimitata** – lunghezza infinita
     Il mittente non si blocca mai
- Nel caso 1. si parla di <u>bufferizzazione esplicita o assente</u>, negli altri due casi si dice che la <u>bufferizzazione è automatica</u>



Scambio dei messaggi – la bufferizzazione (2)

- Politiche di ordinamento delle code dei messaggi nella mailbox e dei processi in attesa
  - First In, First Out
  - Priorità
  - Scadenza

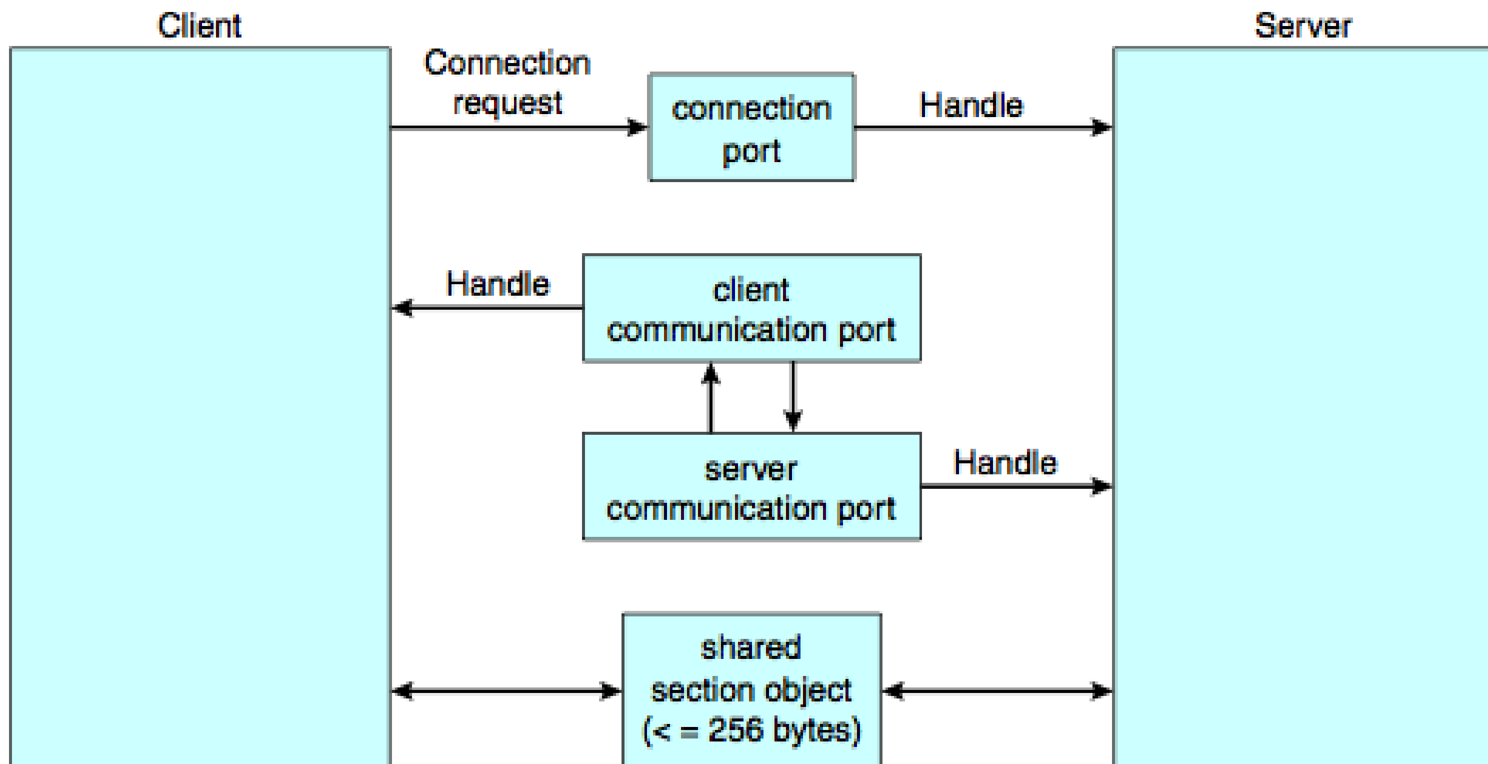
Scambio di messaggi in UNIX

- Vedi codice esempio

Lo scambio dei messaggi in Windows XP (1)

Local Procedure Call

I programmi applicativi sono come client del server di *subsystem* di Windows XP



Lo scambio dei messaggi in Windows XP (2)

- **Dettagli della slide precedente**:
  Windows XP usa due tipi di porte:
  - di connessione (chiamate object) visibili a tutti i processi per instaurare un canale di comunicazione
  - di comunicazione

1. Il client apre la porta di connessione del subsystem con il suo identificatore e manda una richiesta di connessione
2. Il server crea due porte private di comunicazione e restituisce l'identificatore di una di queste al client
3. Il client ed il server usano il corrispettivo identificatore di porta per mandare i msg (memorizzati temporaneamente nella coda dei msg della porta se di lunghezza inferiore a 256 bytes o altrimenti in un *section object* condiviso) o la segnalazione di richiamo (callback), e per ascoltare le risposte

Comunicazione mediante Pipe

- Altri modelli di comunicazione riconducibili al modello "memoria condivisa" sono le pipe
- Una pipe è un tipo speciale di file condiviso
- Ne analizziamo due tipi
  - Convenzionali (o anonime)
  - Named pipe

Pipe convenzionali (1)

- **Comunicazione** tra due processi secondo la modalità del produttore e consumatore
  - Il produttore scrive da un'estremità (**write-end**)
  - Il consumatore legge dall'altra estremità (**read-end**)
- Fa fruire un *flusso di dati* da un processo ad un altro in modo **unidirezionale**

Pipe convenzionali (2)

- Implementata come file in memoria centrale con scrittura solo in aggiunta e lettura unica solo sequenziale
- Una coppia di file descriptor, uno per scrivere e uno per leggere
- Deve esistere una *relazione padre-figlio* tra i due processi per condividere i descrittori di file
- In Unix può essere creata con la chiamata pipe(int fd[])
  - fd è il descrittore del file, fd[0] scrittura, fd[1] lettura
  - Lettura e scrittura tramite le chiamata read() e write()
- *CreatePipe()* in Win32

Pipe convenzionali in Unix (1)

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/types.h>
4.  #include <string.h>

5.  #define BUFFER_SIZE 25
6.  #define READ_END        0
7.  #define WRITE_END       1

8.  int main(void)
9.  {
10.     char write_msg[BUFFER_SIZE] = "Greetings";
11.     char read_msg[BUFFER_SIZE];
12.     pid_t pid;
13.     int fd[2];

14.     /** create the pipe */
15.     if (pipe(fd) == -1) {
16.             fprintf(stderr,"Pipe failed");
17.             return 1;
18.     }

19.     /** now fork a child process */
20.     pid = fork();
```

Pipe convenzionali in Unix (2)

```
21.        if (pid < 0) {
22.        fprintf(stderr, "Fork failed");
23.        return 1;
24.        }
25.        if (pid > 0) {  /* parent process */
26.        /* close the unused end of the pipe */
27.        close(fd[READ_END]);
28.        /* write to the pipe */
29.        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
30.                /* close the write end of the pipe */
31.                close(fd[WRITE_END]);
32.        }
33.        else { /* child process */
34.                /* close the unused end of the pipe */
35.                close(fd[WRITE_END]);
36.                /* read from the pipe */
37.                read(fd[READ_END], read_msg, BUFFER_SIZE);
38.                printf("child read %s\n",read_msg);
39.                /* close the write end of the pipe */
40.                close(fd[READ_END]);
41.        }
42.        return 0;
43.}
```

Named Pipe (1)

- Bidirezionali
- Relazione parentela padre-figlio non necessaria
- Comunicazione fra più di due processi
- Continuano ad esistere anche dopo che i processi comunicanti terminano

Named Pipe (2)

- **In Unix:**
  - Dette FIFO e create con *mkfifo()*, ma sono normali file del file system
  - *Half-duplex* (i dati viaggiano in un'unica direzione alla volta)
  - I programmi comunicanti devono risiedere nella stessa macchina (in alternativa, occorrono i *socket*)
  - Dati byte-oriented
- **In Win32:**
  - Meccanismo più ricco: *createNamedPipe()* e *ConnectNamedPipe()*
  - *Full-duplex* (i dati viaggiano contemporaneamente in entrambe le direzioni)
  - I programmi comunicanti possono risiedere in macchine diverse (ambiente client/server)
  - Dati byte-oriented e message-oriented

Comunicazione mediante file/Pipe

- **Caratteristiche e problemi**:
  - Ordinamento dei messaggi
    - nel file: dipende dal processo scrivente
    - nella pipe: FIFO
  - Ordinamento dei processi in attesa
    - nel file: dipende dalla gestione del file system
  - Sincronizzazione dei processi in lettura e scrittura effettuata secondo politiche e meccanismi del file system

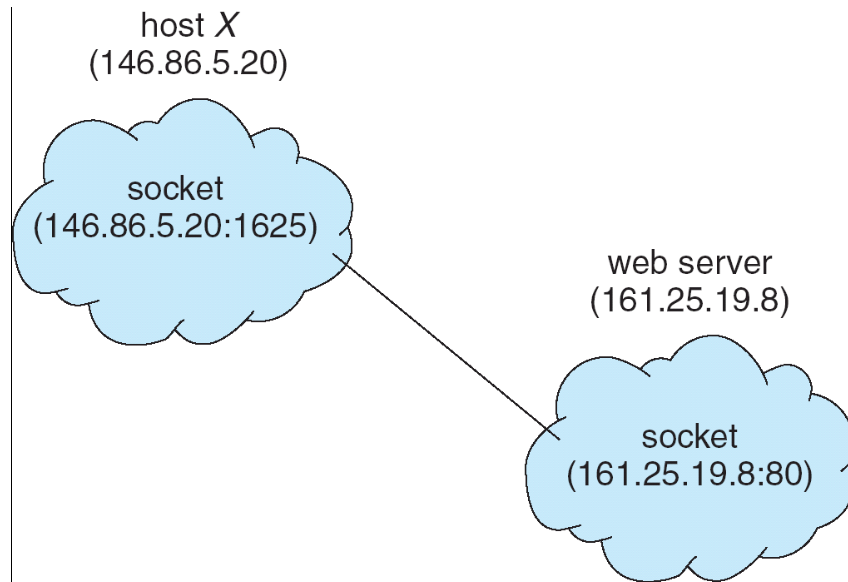
Comunicazione in ambito distribuito

- Socket
- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI) -- in Java

(Non in programma):

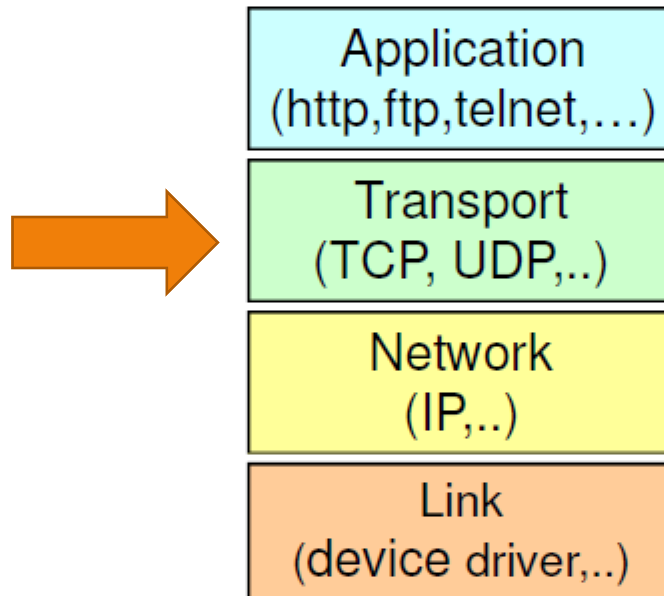
- Protocolli di comunicazione per *applicazioni orientate ai servizi*: SOA/SOAP (Simple Object Access Protocol), http/REST (REpresentation State Transfer), ecc..
- Protocolli di *Internet of Things* (IoT): MQTT (Message Queue Telemetry Transport), AMQP (Advanced Message Queuing Protocol), XMPP, DDS, STOMP, ecc..

Socket



- Un socket è definito come *un estremo di un canale di comunicazione*
- Concatenazione di numero di IP e numero di porta
- Il socket **161.25.19.8:1625 è assegnato alla porta 1625 sul** server **161.25.19.8**
- La connessione consisterà in una coppia di socket
- I server che implementano specifici servizi (ftp, http, ecc..) ascoltano porte note (ftp su 21, http su 80, ecc..)
  - tutte le porte sotto la 1024 sono considerate *note*

Comunicazione via socket



I socket possono essere orientati alla connessione (TCP) e senza connessione (UDP)

Servizi con connessione

- **Servizi con connessione (connection oriented)**
  - Il servizio è offerto attraverso **tre fasi**:
    1. **Apertura** della connessione tra due punti della rete
    2. **Utilizzo** della connessione per inviare i dati
    3. **Chiusura** della connessione
  - Caratteristica fondamentale: i dati sono ricevuti nello stesso ordine in cui vengono inviati
  - Analogia: sistema telefonico
  - Due varianti: stream di messaggi e stream di byte

Servizi senza connessione

- **Servizi senza connessione (connection-less)**
  - Non c'è alcuna connessione
  - I dati sono inviati impacchettati in messaggi, ognuno dei quali contiene l'indirizzo completo del destinatario
  - I messaggi non arrivano necessariamente nell'ordine in cui sono inviati
  - Analogia: sistema postale

Applicazione	**Protocollo strato applicazione**	**Protocollo strato trasporto**
Posta elettronica	SMTP	TCP
Accesso a terminale remoto	telnet	TCP
Trasferimento file	FTP	TCP
Web	HTTP	TCP
Streaming Audio/Video	RTSP/RTP	TCP (comandi) + UDP (flusso)
Server di file remoto	NFS	tipicamente UDP
Telefonia su internet (VoIP)	SIP, H.323, altri	tipicamente UDP
Gestione della rete	SNMP	tipicamente UDP
Protocollo di routing	RIP	tipicamente UDP
Risoluzione dei nomi	DNS	tipicamente UDP

Comunicazione via socket in Java

Il package **Java.net** fornisce due classi **per i socket TCP:**

- ServerSocket – per implementare un programma server
- Socket – per implementare un programma client

Per i **socket UDP**:

- DatagramSocket
- MulticastSocket: per socket *multicast* (destinatari multipli)

Classi Java per gli *stream di I/O*:

- **OutpuStream/InputStream** per stream di byte
- **Writer/Reader** per stream di caratteri
- **PrintStream** (ad es. System.out) aggiunge funzionalità ad uno stream di output; non solleva eccezioni
- **DataInputStream/DataOutputstream** per leggere tipi primitivi da uno stream di byte
- **ObjectInputStream, ObjectOutputstream** per inviare oggetti, o meglio serializzare/deserializzare oggetti in byte – metodi writeObject(o) / readObject())

Comunicazione via socket TCP in Java (1)

```
public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // now listen for connections
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Il Server restituisce una data

La chiamata al metodo accept() è bloccante

Comunicazione via socket TCP in Java (2)

```
public class DateClient
{
   public static void main(String[] args) {
      try {
         //make connection to server socket
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         // read the date from the socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // close the socket connection
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Socket: vantaggi e svantaggi

- Semplice
- Efficiente
- Di basso livello (livello di trasporto): permette la trasmissione di un flusso non strutturato di byte
  - È responsabilità di Client e Server interpretare e organizzare i dati in forme complesse
  - RPC e RMI risolvono questo problema

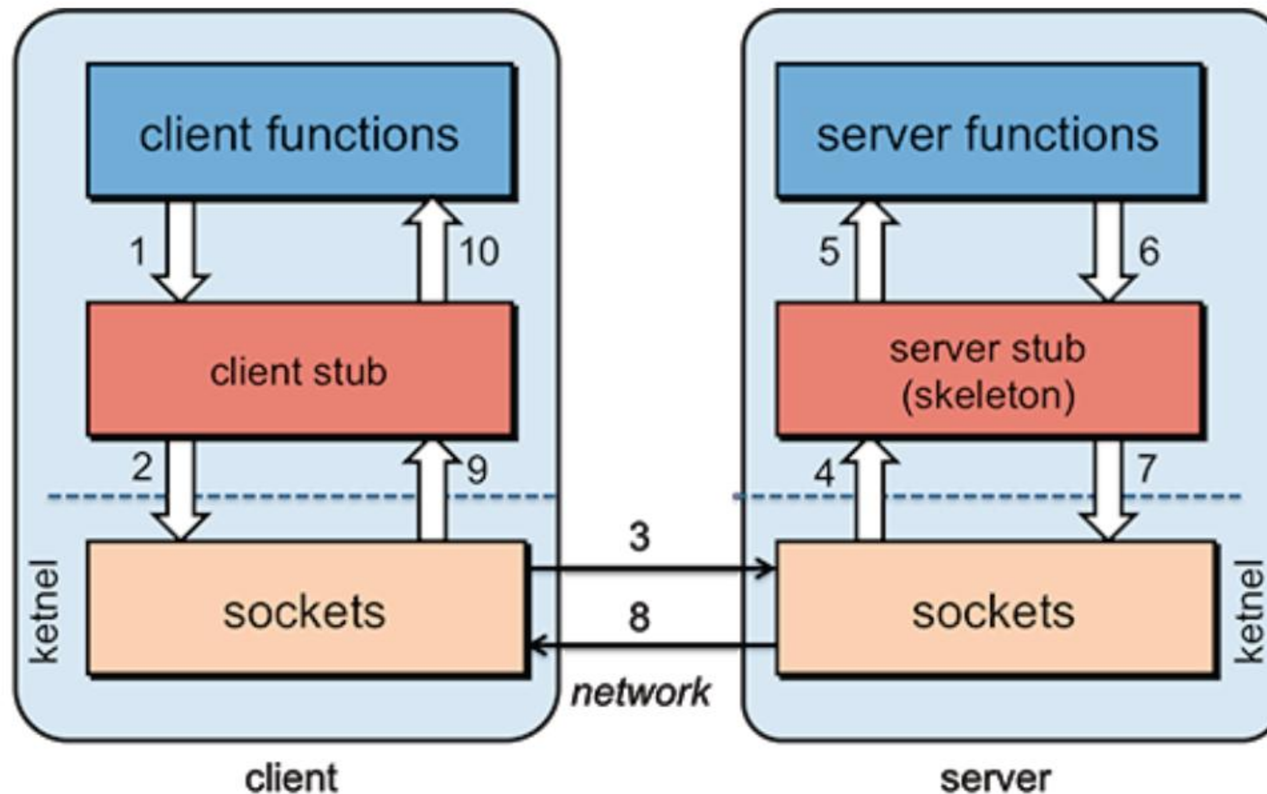
Remote Procedure Call (RPC) (1)

- La chiamata di procedura remota (RPC) astrae il meccanismo di chiamata di procedura per usarlo fra sistemi con una connessione di rete
- Il Client può invocare una procedura remota nello stesso modo in cui ne invocherebbe una locale
- Il server ha una porta per ogni RPC
- RPC nasconde i dettagli della comunicazione assegnando al client uno **stub** (uno per ogni procedura remota)
  - Stub: segmento di codice che permette di invocare la procedura remota

Remote Procedure Call (RPC) (2)

- Il client invoca una procedura remota passando i parametri allo stub
- Lo **stub lato client** esegue il **marshalling** dei parametri e li trasmette al server usando tecniche di scambio di messaggi
  - Marshalling : strutturazione dei parametri in un formato che può essere trasmesso via rete
  - Necessario per via dell'uso di strutture dati complesse e differente rappresentazione lato client/server dei dati, ad es. 32 vs 64 bit (*little-endian vs big-endian*)
- Nel server un analogo del client, lo **skeleton**, riceve la chiamata di procedura, spacchetta i parametri tradotti e invoca la procedura stessa e (se necessario) restituisce il risultato al client
  - Eventuali valori di ritorno sono passati al client con la stessa tecnica

Remote Procedure Call (RPC) (3)



Remote Procedure Call (RPC) (4)

- Un problema è la **semantica della chiamata**
  - Le chiamate RPC possono fallire o essere duplicate ed eseguite più volte, come risultato di problemi sulla rete

- Due possibili interpretazioni:
  - Il SO può assicurare che la chiamata venga eseguita
    a) *esattamente una volta* o
    b) *al più una volta*

Remote Procedure Call (RPC) (5)

- La semantica (a) "*al più una volta*" è garantita associando a ciascun msg una marca di tempo; il server mantiene uno storico delle marche delle chiamate già eseguite
  - Se riceve una chiamata già eseguita la scarta

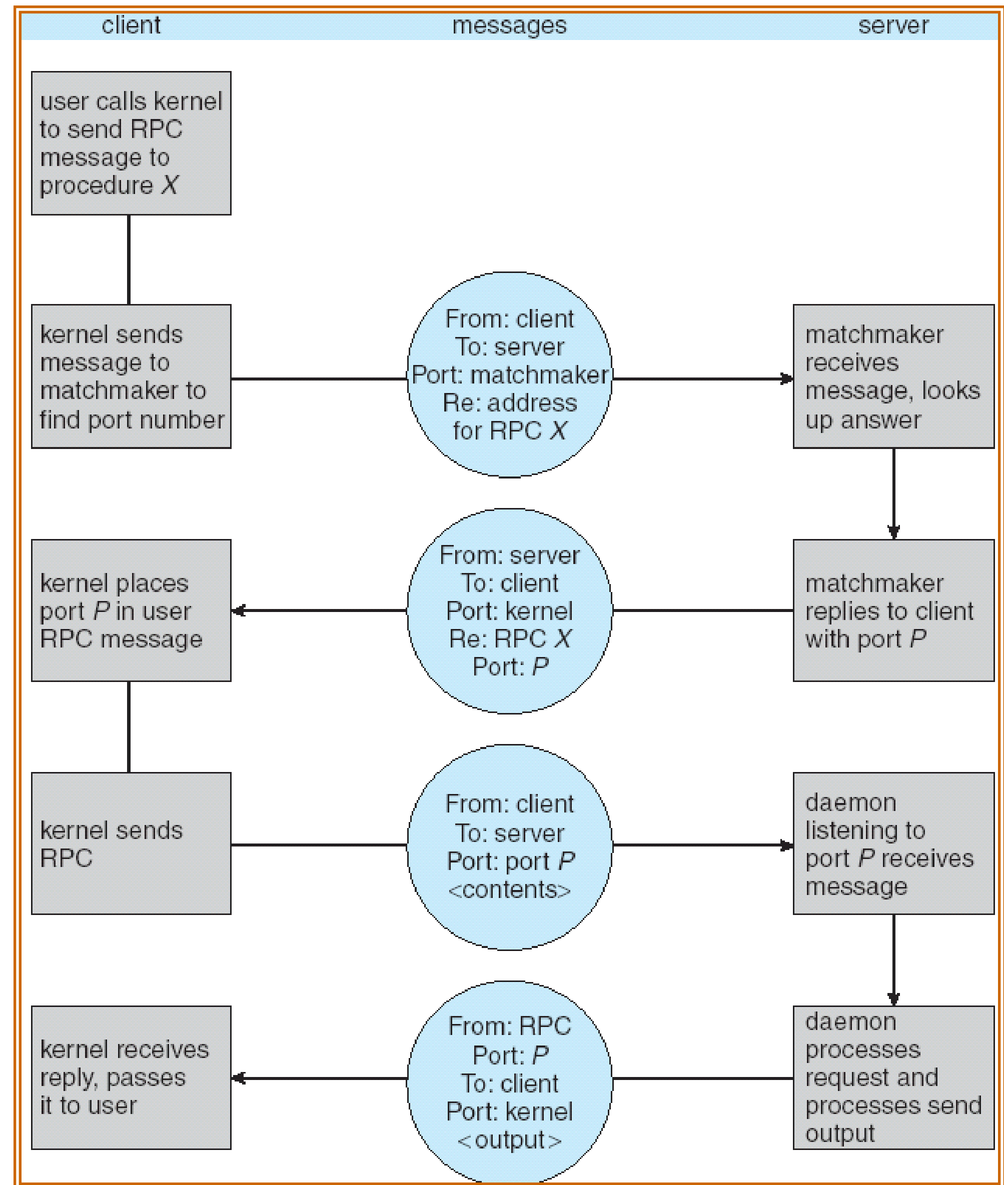
- La semantica (b) "*esattamente una volta*" garantisce l'esecuzione della chiamata
  - Il server implementa il protocollo (a) e in più notifica al client che la chiamata RPC è stata ricevuta ed eseguita (msg ACK di acknowledgement)
  - Il client deve rispedire ciascuna chiamata RPC periodicamente fino a quando non riceve un ACK per ogni chiamata

Remote Procedure Call (RPC) (6)

Un ultimo problema riguarda l'**associazione Client-Server**

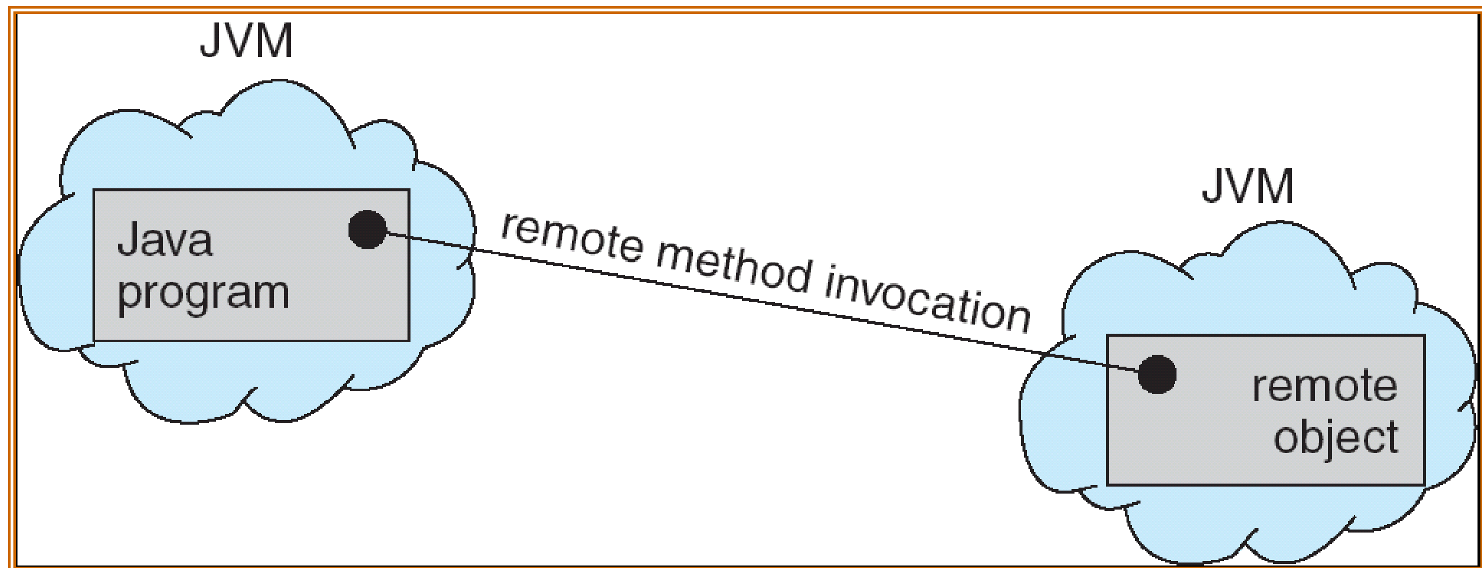
Come può il client conoscere il n. di porta del server?

1. *tramite indirizzo prefissato:* *a*ssociazione fissa e nota RPC-Porta
2. *dinamicamente* mediante un servizio del SO di rendezvous (o matchmaker -- accoppiatore): il server ha un demone in ascolto che riceve una richiesta dal client e restituisce il numero di porta prefissato sul server, <u>come illustrato in figura</u>

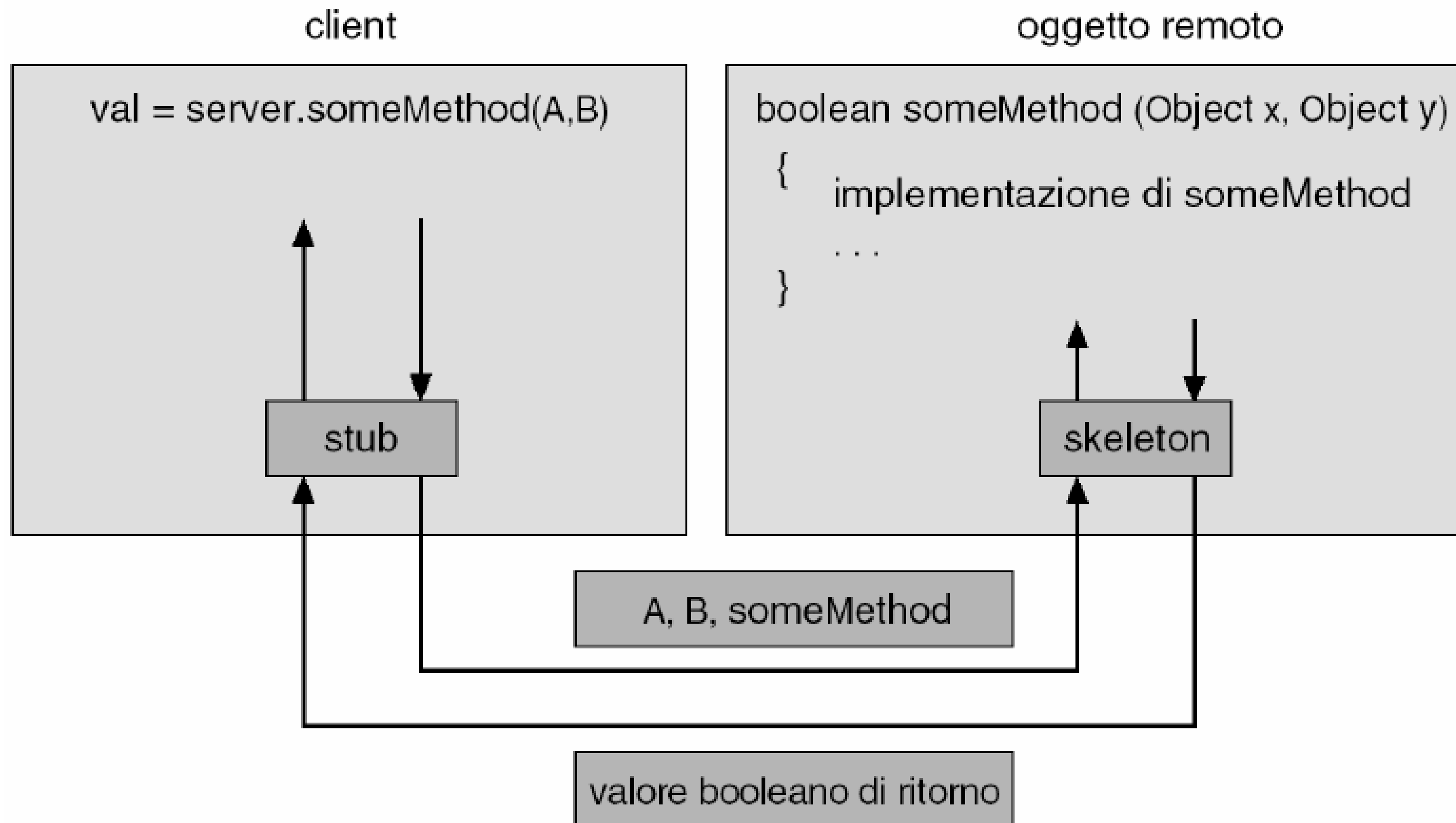


Remote Method Invocation (RMI)

- L'invocazione di metodo remoto è una caratteristica di Java simile alla RPC
- RMI permette ad un programma Java di invocare metodi su oggetti remoti
- I parametri in RPC sono strutture dati ordinarie, mentre in RMI sono oggetti!



RMI: Marshalling dei parametri (1)



RMI: Marshalling dei parametri (2)

Se i parametri sono oggetti:

- Se sono **oggetti locali**, essi vengono passati per copia tramite una tecnica nota come ***serializzazione***
  - lo stato di un oggetto è scritto in uno stream di byte
    - si serializzano ricorsivamente i membri dell'oggetto
    - tipi semplici sono mappati direttamente
  - Molti oggetti delle API Java sono serializzabili (ovvero implementano l'interfaccia java.io.Serializable)

- Se sono **oggetti remoti**, vengono passati per riferimento (*remote reference*)
  E' un oggetto remoto un oggetto che estende l'interfaccia **java.rmi.Remote**

RMI Example

```
public interface RemoteDate extends Remote
{
   public abstract Date getDate() throws RemoteException;
}
```

RMI Example

```
public class RMIClient
{
   public static void main(String args[]) {
      try {
         String host = "rmi://127.0.0.1/DateServer";

         RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
         System.out.println(dateServer.getDate());
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

RMI Example

La classe UnicastRemoteObject estende RemoteServer e fornisce i metodi per implementare un server RMI

La classe Naming fornisce metodi statici per memorizzare (metodo rebind slide precedente) e ottenere (metodo lookup) un riferimento ad un oggetto remoto in un registro remoto (nameserver)

```
public class RemoteDateImpl extends UnicastRemoteObject
      implements RemoteDate
{
   public RemoteDateImpl() throws RemoteException { }

   public Date getDate() throws RemoteException {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         RemoteDate dateServer = new RemoteDateImpl();

         // Bind this object instance to the name "DateServer"
         Naming.rebind("DateServer", dateServer);
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```