SISTEMI OPERATIVI

(MODULO DI INFORMATICA II)

La comunicazione tra processi

Prof. Luca Gherardi

Prof.ssa Patrizia Scandurra (anni precedenti)

Università degli Studi di Bergamo

a.a. 2012-13

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 ambienti client-server

Dipendeza tra processi

- Un **processo indipendente** non può influenzare o essere influenzato dagli altri processi in esecuzione
  - Significa che **non condivide** risorse con altri processi

- Un **processo cooperante** può influenzare o essere influenzato da altri processi in esecuzione nel sistema
  - Significa che **condivide** risorse con altri processi

Processi cooperanti

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

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

Cooperazione

Cooperazione = lavoro congiunto di processi per raggiungere scopi applicativi comuni con condivisione e scambio di informazioni

Vantaggi del processo di cooperazione:

- Condivisione delle informazioni (ad es. un file o una directory condivisa)
  - Accesso concorrente
- Velocizzazione della computazione (possibile in verità solo con più CPU o canali di I/O)
  - Esecuzione di sotto-attività in parallelo
- Modularità (dividendo le funzioni del sistema in processi o thread separati)
  - Esempio sistemi operativi modulari
- Convenienza (un singolo utente può lavorare su molte attività)
  - Multi tasking

Esempi

- Processi cooperanti
  - Client - Server
  - Compilatore - Assemblatore – Loader
  - Produttore - Consumatore

Comunicazione

Inter-Process Communication (IPC)

- Meccanismi e politiche
  - che permetto ai processi di scambiarsi informazioni
  - per operare in modo cooperativo

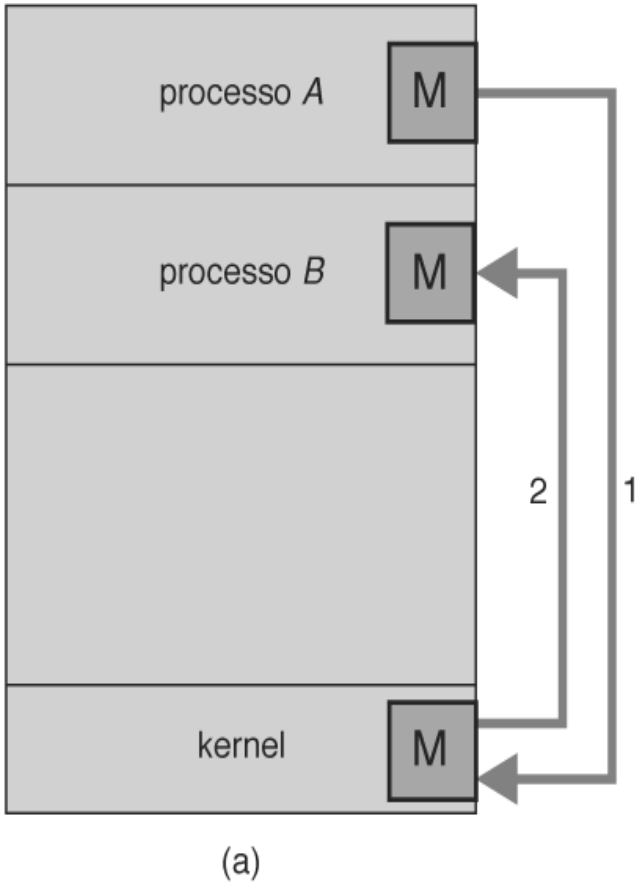
- Necessità
  - Trasferimento di informazioni da processo mittente a ricevente
  - Condivisione di informazioni

IPC: Caratteristiche

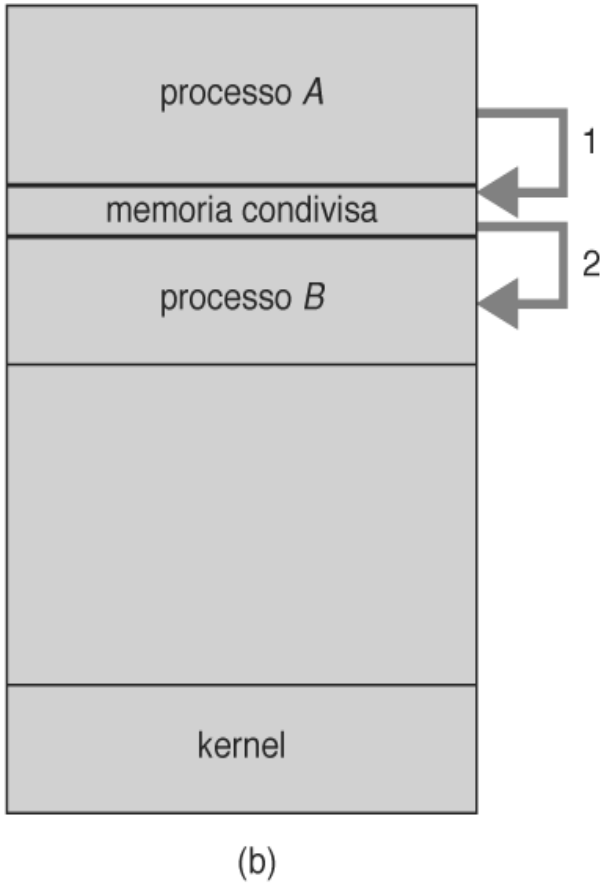
- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità
- Semplicità di uso nelle applicazioni
- Omogeneità delle comunicazioni
- Integrazione nel linguaggio di programmazione
- Affidabilità
- Sicurezza
- Protezione

IPC: Implementazione

Scambio di messaggi



Memoria condivisa



MEMORIA CONDIVISA

Memoria condivisa

- I processi comunicanti colloquiano attraverso un'area di memoria condivisa
  - Tipicamente residente nello spazio degli indirizzi di chi la alloca
  - Gli altri processi annettono questa area al loro spazio degli indirizzi
- Tipicamente i processi non possono accedere alla memoria degli altri
  - I processi comunicanti devono quindi stabilire un "accordo"
- Il SO non controlla
  - Tipo e allocazione dei dati
  - Accesso concorrente alla memoria

Memoria condivisa

- Caratteristiche:
  - Condivisione variabili globali
  - 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* (meccanismi che vedremo più avanti)

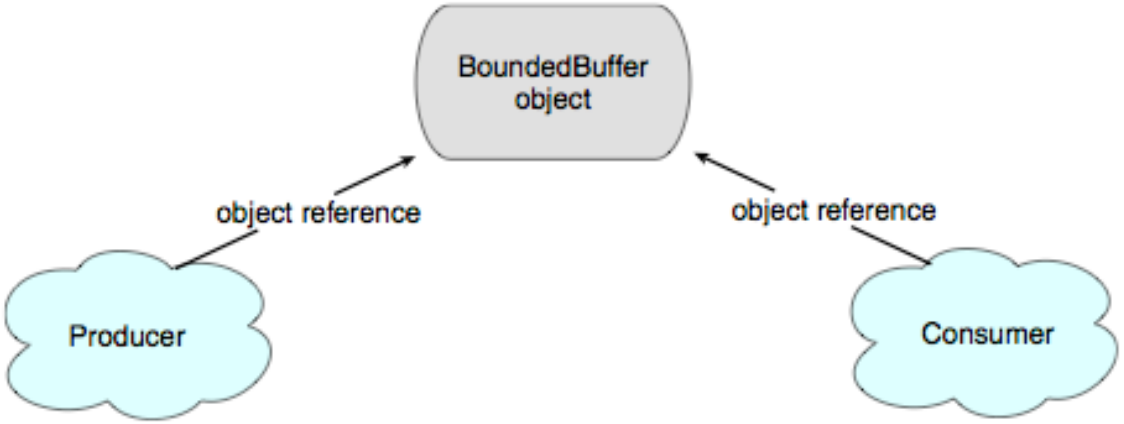
Il problema del produttore-consumatore (1)

- Modello molto comune nei processi cooperanti:
  - un processo **produttore** genera informazioni
  - che sono utilizzate da un processo **consumatore**
- Un oggetto temporaneo in memoria (buffer) può essere riempito dal produttore e svuotato dal consumatore

1. **buffer illimitato (unbounded-buffer)**: non c'è un limite teorico alla dimensione del buffer
   - Il consumatore deve aspettare se il buffer è vuoto
   - Il produttore può sempre produrre
2. **buffer limitato (bounded-buffer)**: la dimensione del buffer è fissata
   - Il consumatore deve aspettare se il buffer è vuoto
   - Il produttore deve aspettare se il buffet è pieno

Il problema del produttore-consumatore (2)

- Il buffer può essere
  - fornito dal SO tramite l'uso di funzionalità di comunicazione tra processi **InterProcess Communication (IPC)**
  - oppure essere esplicitamente scritto dal programmatore dell'applicazione con l'uso di **memoria condivisa**
- Simuliamo in Java il problema permettendo ai processi di condividere il buffer http://www.doc.ic.ac.uk/~jnm/book/book_applets/BoundedBuffer.html



Buffer limitato – Interfaccia per le implementazioni del buffer in Java

```
public interface Buffer
{
    //  i produttori chiamano questo metodo
    public abstract void insert(Object item);

    // i consumatori chiamano questo metodo
    public abstract Object remove();
}
```

Bounded-buffer in Java – memoria condivisa

```
import  java.util.*;

public  class  BoundedBuffer  implements  Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count; // numero di elementi nel buffer
    private int in; // punta alla successiva posizione libera
    private int out; // punta alla successiva posizione piena
    private Object[] buffer;

    public BoundedBuffer() {
        // il buffer è inizialmente vuoto
        count = 0;
        in = 0;
        out = 0;

        buffer = new Object[BUFFER_SIZE];
    }

    //i produttori chiamano questo metodo
    public void insert(Object item) {
      vedi slide successiva
    }

    // i consumatori chiamano questo metodo
    public Object remove() {
      vedi slide successiva
    }

}
```

Bounded-buffer in Java – memoria condivisa: insert()

```
public void insert(Object item) {
    while (count == BUFFER_SIZE)
        ; //  non fare nulla - non ci sono buffer liber

    // aggiungi un elemento al buffer
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-buffer in Java – memoria condivisa: remove()

```
public Object remove() {
    Object item;

    while (count == 0)
        ; // non fare nulla - nulla da consumare

    // rimuovi un elemento dal buffer
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    return item;
}
```

Uso della 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

Uso della 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 DI MESSAGGI

Scambio dei messaggi

- **Caratteristiche**
  - L'uso della 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

- **Problemi**
  - Sincronizzazione per l'accesso ai messaggi
  - gestita però implicitamente dal SO fornendo due operazioni:
    - **send** (messaggio)
    - **receive** (messaggio)

Scambio dei messaggi – I messaggi

- Contenuto messaggio
  - Processo mittente
  - Processo destinatario
  - Informazioni da trasmettere
  - Eventuali altre informazioni di gestione dello scambio messaggi
- Dimensione messaggio
  - Fissa
    - Facile implementazione a livello SO, difficile a livello applicazione
  - Variabile
    - Difficile implementazione a livello SO, facile a livello applicazione

Scambio dei messaggi – I 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)

Tre tematiche importanti:

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

La comunicazione diretta simmetrica

- 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

La comunicazione diretta asimmetrica

- Solo il processo che invia il messaggio deve conoscere **esplicitamente** il nome del destinatario o del mittente:
  - **send (P, msg)** – manda un messaggio al processo P
  - **receive (id, msg)** – riceve un messaggio da un processo il cui identificatore è salvato in id

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** (M, msg) – manda un messaggio alla mailbox M
  - **receive** (M, msg) – riceve un messaggio dalla mailbox M

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

La comunicazione indiretta (3)

- Una mailbox può appartenere ad un processo o al SO
- In ogni caso esiste sempre un processo che ha il diritto di proprietà

- Se appartiene al processo allora risiede nel suo spazio di indirizzi
  - Viene deallocata alla terminazione del processo
  - Il proprietario è l'unico che può ricevere
  - Tutti gli altri sono **utenti** che inviano messaggi

La comunicazione indiretta (4)

- Per le mailbox appartenenti al SO, il SO stesso mette a disposizione delle chiamate di sistema per:
  - Creare mailbox
  - Cancellare mailbox
  - Inviare e ricevere messaggi
  - Cedere o concedere il diritto di proprietà sulla mailbox
- Il processo creatore ha il **diritto di proprietà**
  - Inizialmente è l'unico che può ricevere
- La concessione del diritto di proprietà può portare ad avere più riceventi

La comunicazione indiretta (5)

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 o tramite algoritmo quale processo riceverà il messaggio. Il sistema può anche notificare il ricevente al mittente

La sincronizzazione

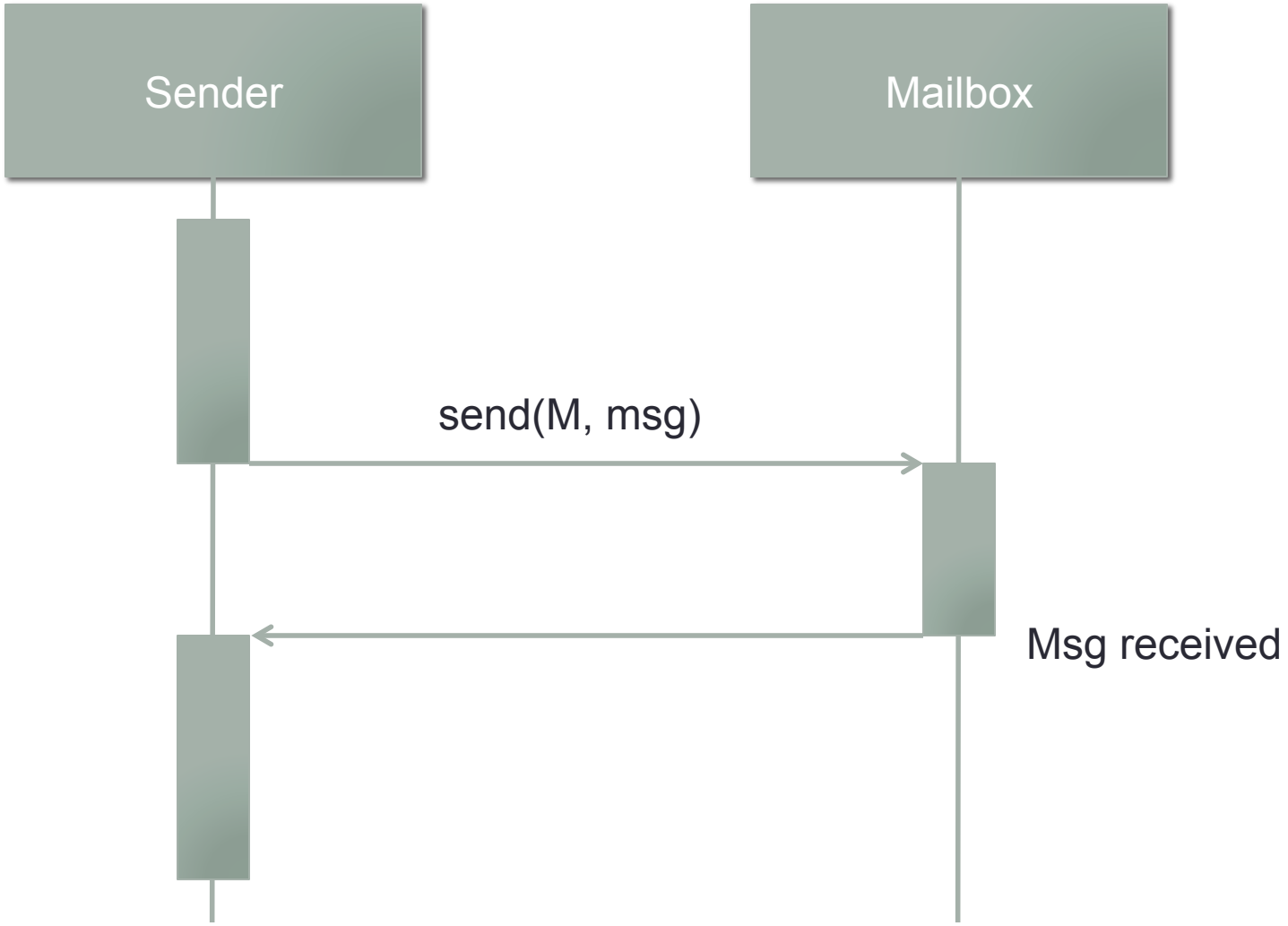
- Il passaggio di messaggi può essere bloccante (sincrono) oppure non bloccante (asincrono)

RENDEZVOUS

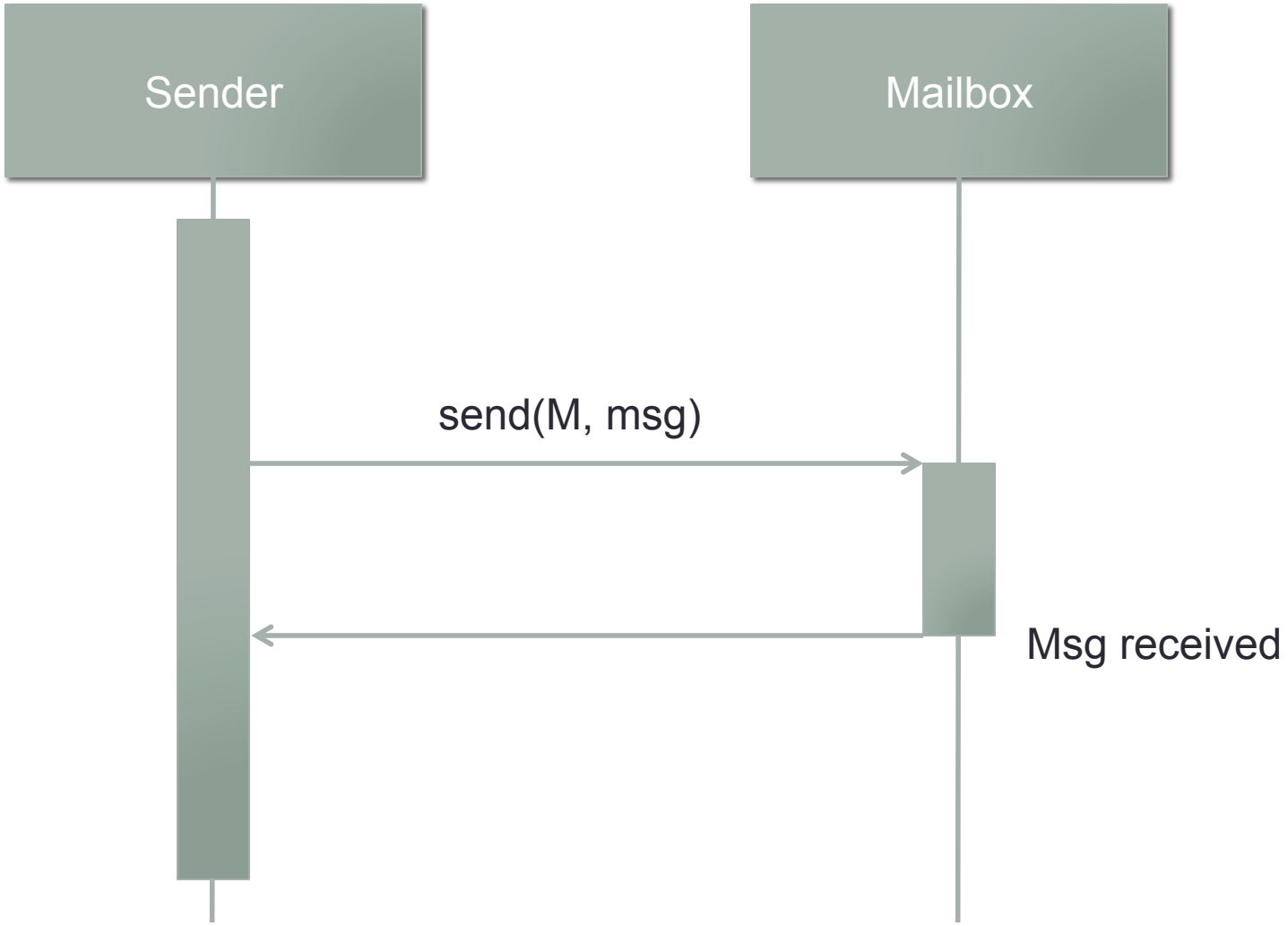
  - **Invio bloccante**: il processo che invia viene bloccato finché il messaggio viene ricevuto dal processo che riceve o dalla mailbox
  - **Ricezione bloccante**: il ricevente si blocca sin quando un messaggio non è disponibile

  - **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

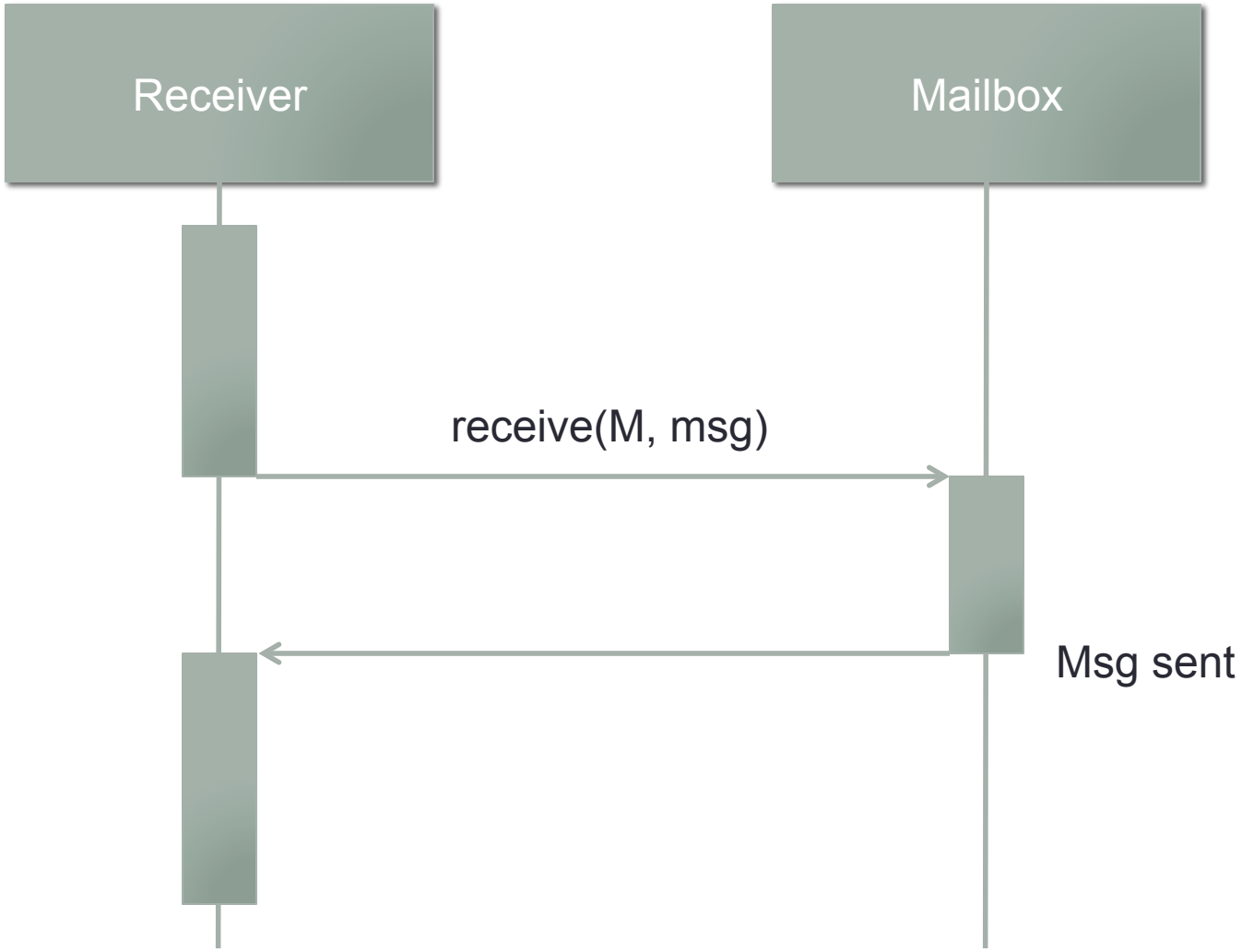
La sincronizzazione - Invio bloccante



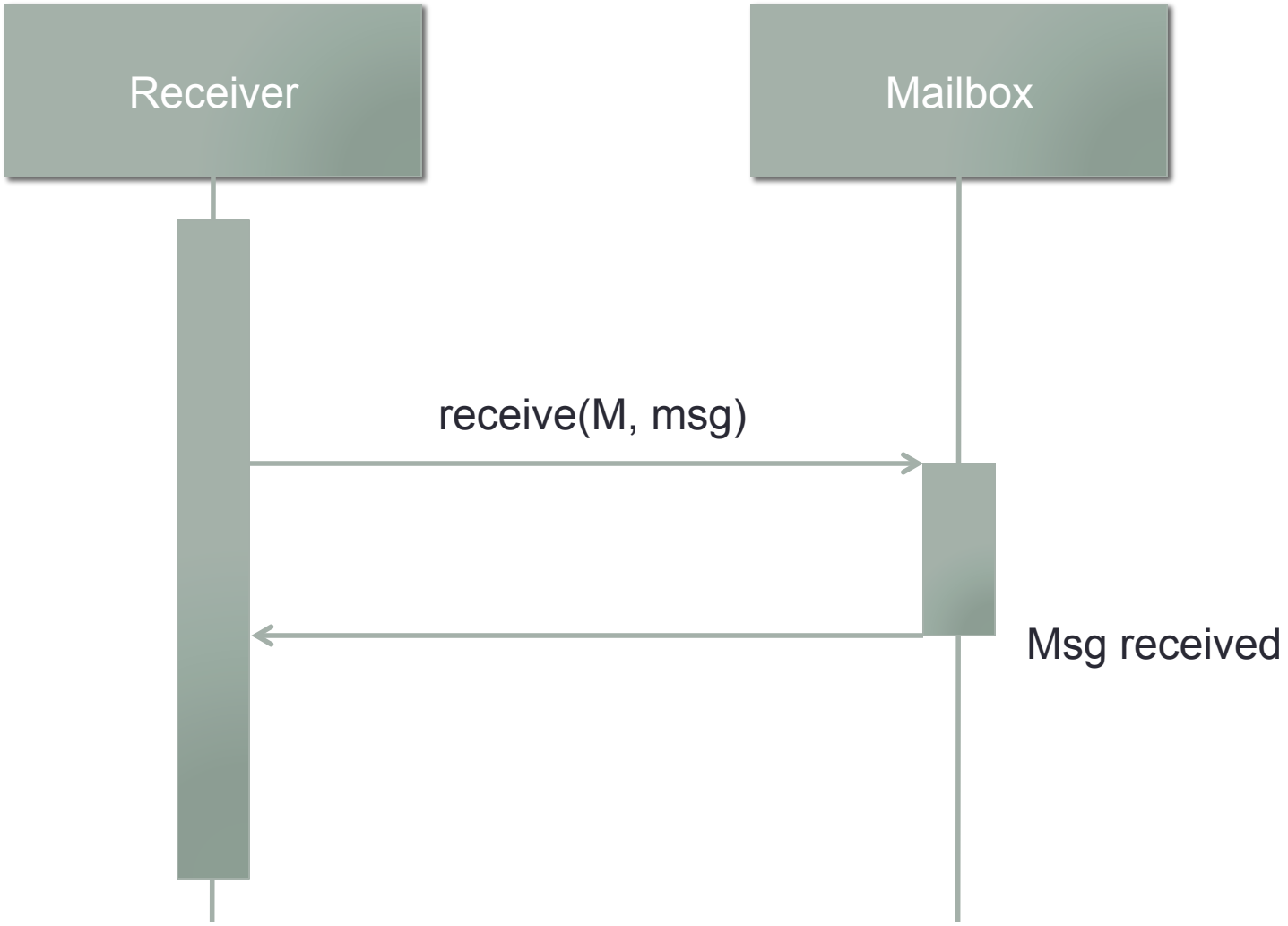
La sincronizzazione - Invio non bloccante



La sincronizzazione - Ricezione bloccante

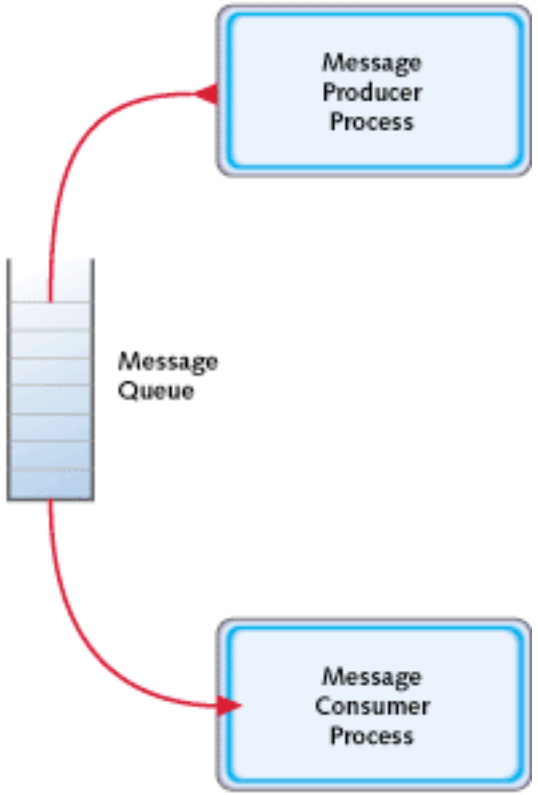


La sincronizzazione - Ricezione non bloccante



La bufferizzazione (1)

- I messaggi scambiati tra processi risiedono in code temporanee (sia comunicazione diretta che indiretta)
- Tre modi per implementare le code:
  1. **Capacità zero** – Il mittente deve bloccarsi finché 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 bufferizzazione esplicita o assente, negli altri due casi si dice che la bufferizzazione è automatica



La bufferizzazione (2)

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

Message Queue in Java – scambio dei messaggi: channel interface

```
public interface Channel
{
   // Send a message to the channel
   public abstract void send(Object item);

   // Receive a message from the channel
   public abstract Object receive();
}
```

Message Queue in Java – scambio dei messaggi: channel implementation

```
public class MessageQueue implements Channel
{
    private Vector queue;

    public MessageQueue() {
       queue = new Vector();
    }

    // This implements a nonblocking send
    public void send(Object item) {
       queue.addElement(item);
    }

    // This implements a nonblocking receive
    public Object receive() {
       if (queue.size() == 0)
          return null;
       else
          return queue.remove(0);
    }
}
```

Message Queue in Java – scambio dei messaggi: produttore

```
Channel mailBox;

while (true) {
    Date message = new Date();
    mailBox.send(message);
}
```

Bounded-buffer in Java – scambio dei messaggi: consumatore

```
Channel mailBox;

while (true) {
   Date message = (Date) mailBox.receive();
   if (message != null)
      // consume the message
}
```

Lo scambio dei messaggi in Windows XP (2)

Windows XP usa due tipi di porte:

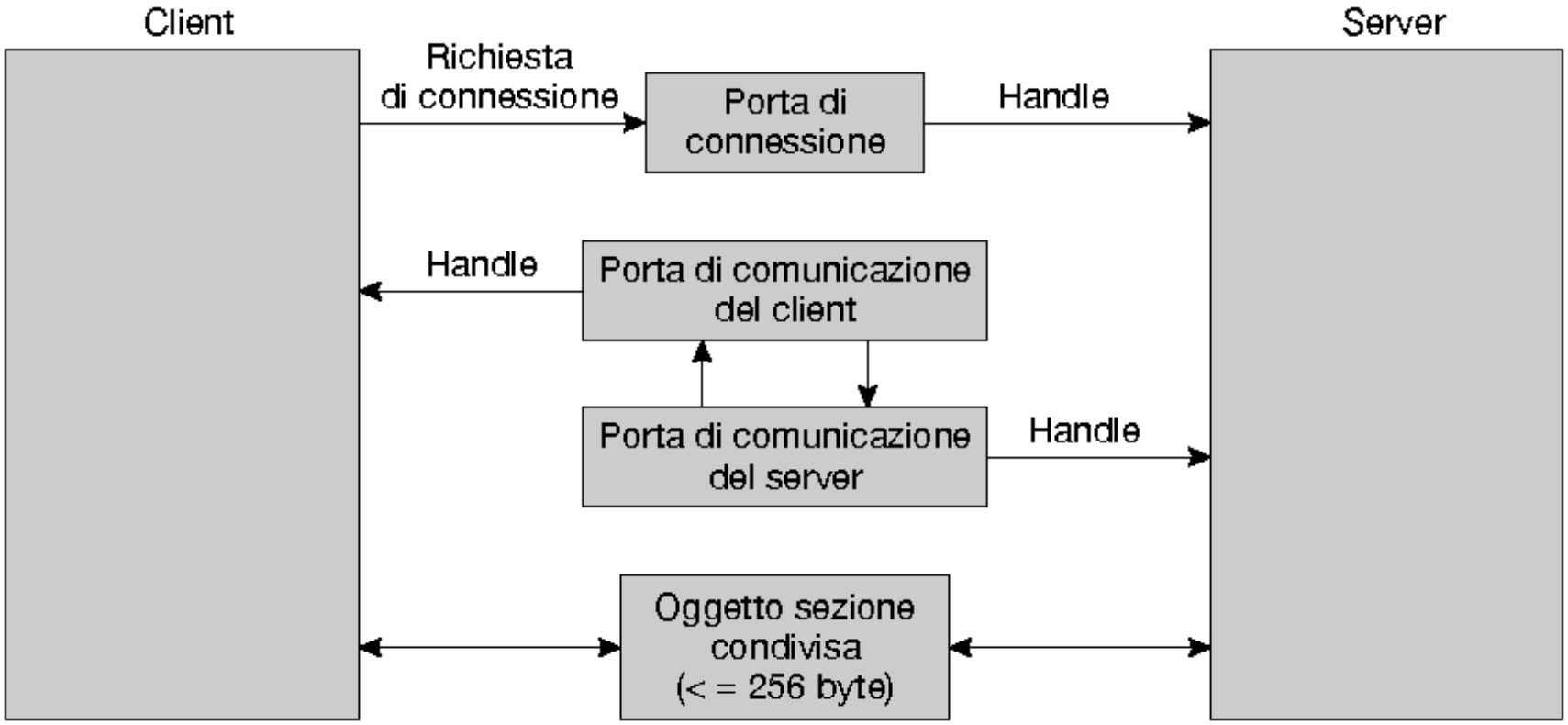
- di connessione (chiamate object) visibili a tutti i processi per instaurare un canale di comunicazione
- di comunicazione

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

Lo scambio dei messaggi in Windows XP (1)

Local Procedure Call

- I programmi applicativi sono come client del server (locali) di *subsystem* di Windows XP



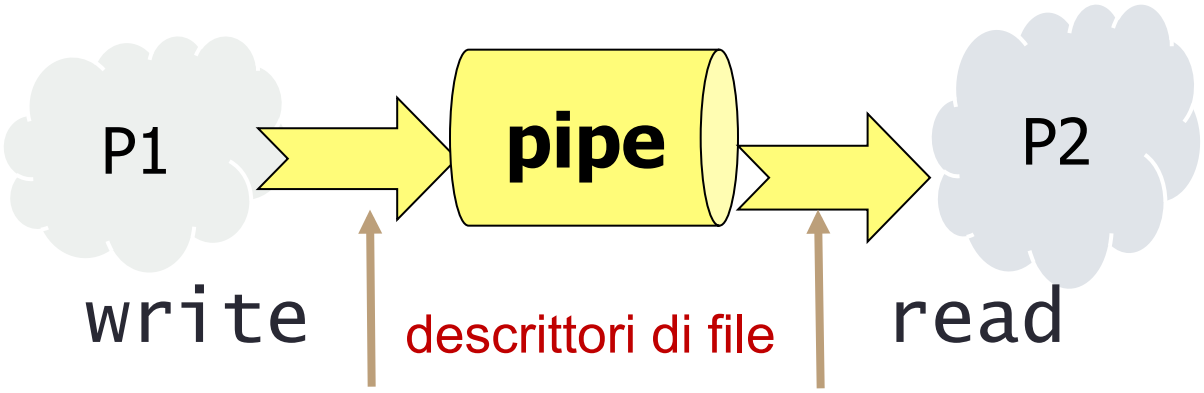
COMUNICAZIONE TRAMITE PIPE

Comunicazione tramite 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 econsumatore
- Il produttore scrive da un'estremità (**write-end**)
- Il consumatore legge dall'altra estremità (**read-end**)
- Comunicazione uni-direzionale



Pipe convenzionali (2)

- Implementata come file in memoria centrale con scrittura solo in aggiunta e lettura unica solo sequenziale
- 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()`

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 (1)

- **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()` (nuova pipe) e `ConnectNamedPipe()` (pipe già esistente)
  - 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 IN AMBIENTI CLIENT-SERVER

Socket

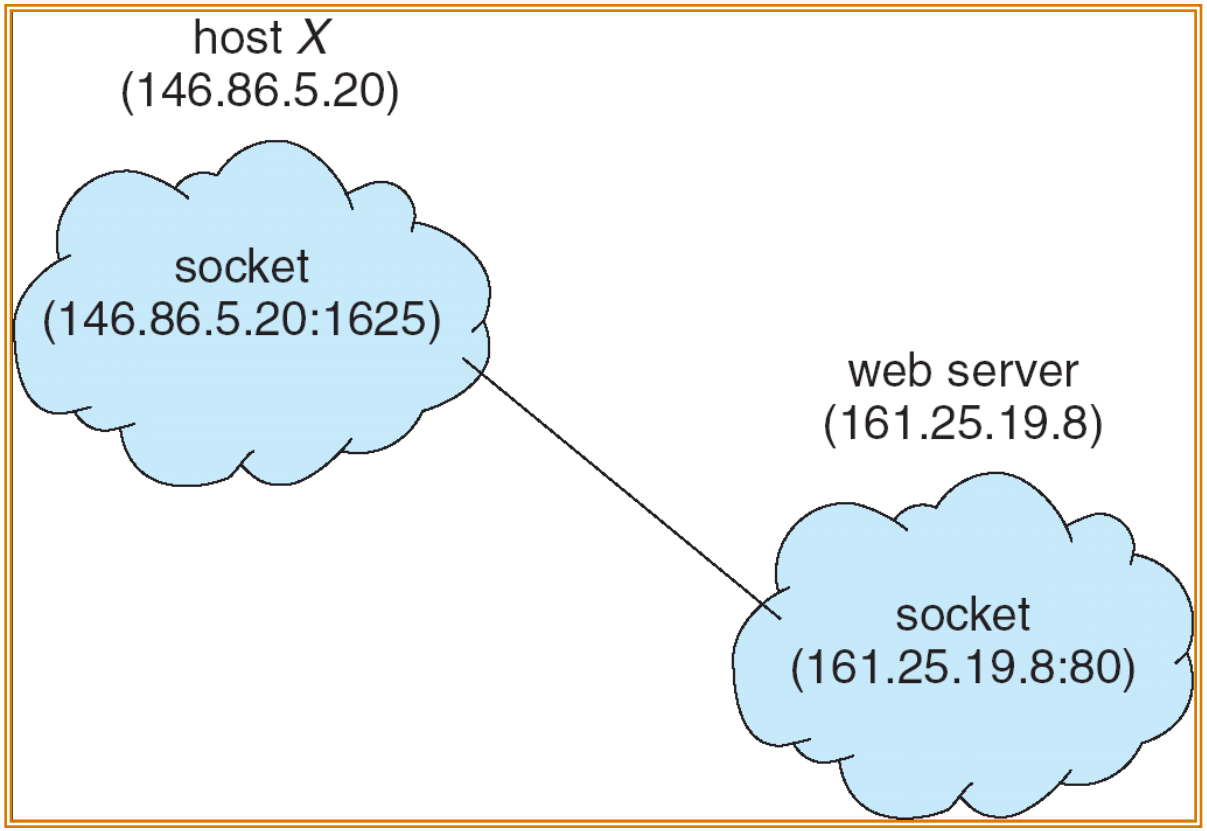
Remote Procedure Calls (RPC)

Remote Method Invocation (RMI)

Socket

- Una socket è definito come **un estremo di un canale di comunicazione**
- **Una copia di processi può comunicare attraverso una copia di socket, una per processo**
- Una socket è identificata da numero di IP e numero di porta
- Il socket **161.25.19.8:1625 è assegnato alla porta 1625 sul** calcolatore **161.25.19.8**
- 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
- Tutte le connessioni devono essere uniche: se un processo vuole due connessioni verso un server deve creare due socket

Comunicazione via socket



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

Comunicazione via socket in Java – Il server

```
import java.net.*;
import java.io.*;

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

      // si pone in ascolto di richieste di connessione
      while (true) {
        Socket client = sock.accept();

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

        // scrive la Data sulla socket
        pout.println(new java.util.Date().toString());

        // chiude la socket e ritorna in ascolto
        // di nuove richieste
        client.close();
      }
  }
  catch (IOException ioe) {
    System.err.println(ioe);
  }
 }
}
```

Il Server restituisce una data

La chiamata al metodo accept() è bloccante

Comunicazione via socket in Java – Il client

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      //si collega alla porta su cui ascolta il server
      Socket sock = new Socket("127.0.0.1",6013);

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

      // legge la data dalla socket
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      // chiude la connessione tramite socket
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Socket – Vantaggi e svantaggi

- Semplice
- Efficiente

- Di basso livello: 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

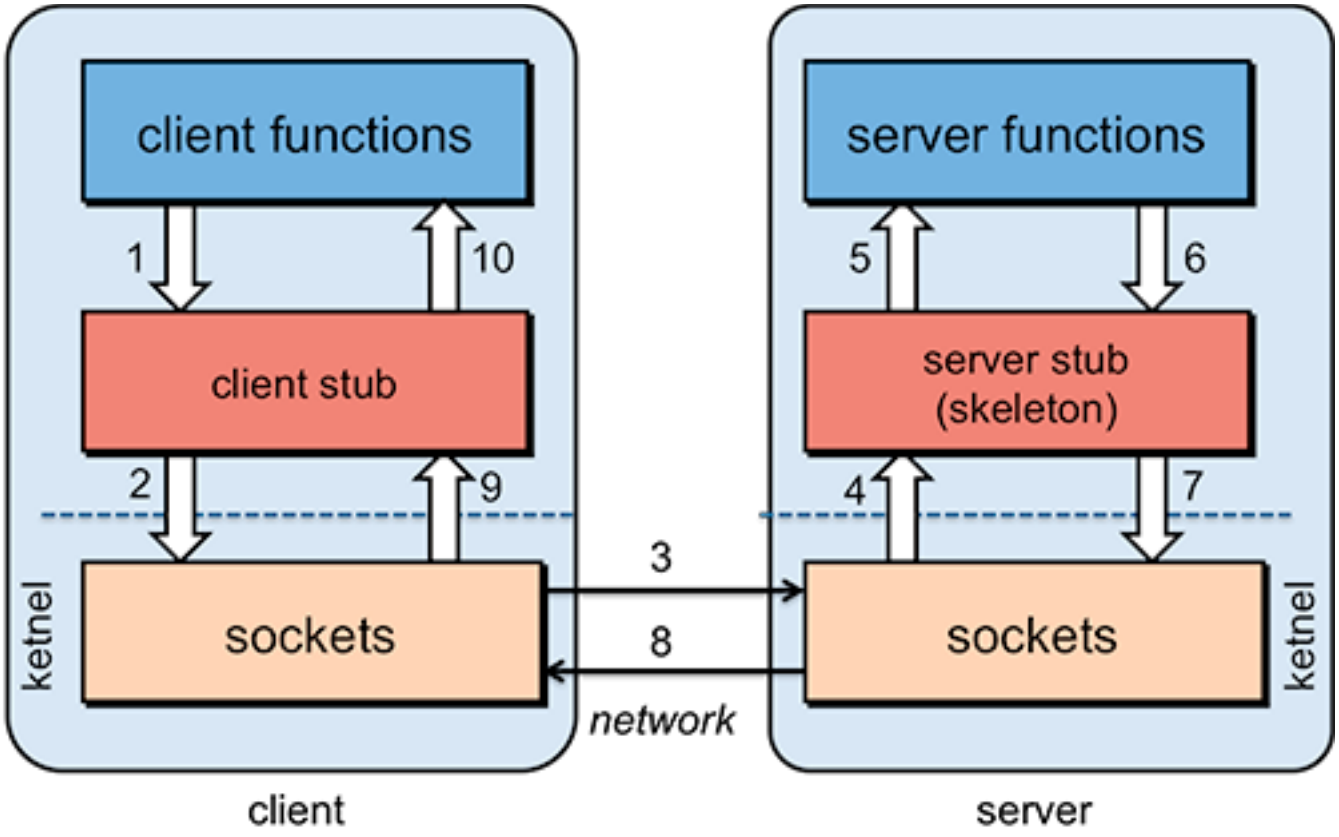
Remote Procedure Call (RPC) (1)

- 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**
  - Stub: segmento di codice che permette di invocare la procedura remota (uno per ogni procedura remota)

Remote Procedure Call (RPC) (2)

- Il client invoca una procedura remota passando i parametri allo stub
- Questo 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 (e.g. 32 vs 64 bit)
- Nel server un analogo del client, lo **skeleton**, riceve la chiamata di procedura, invoca la procedura stessa e (se necessario) restituisce il risultato al client

Remote Procedure Call (RPC) (3)



Remote Procedure Call (RPC) (4)

- Un altro problema (oltre al marshalling) 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) al più una volta, o
    b) esattamente 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

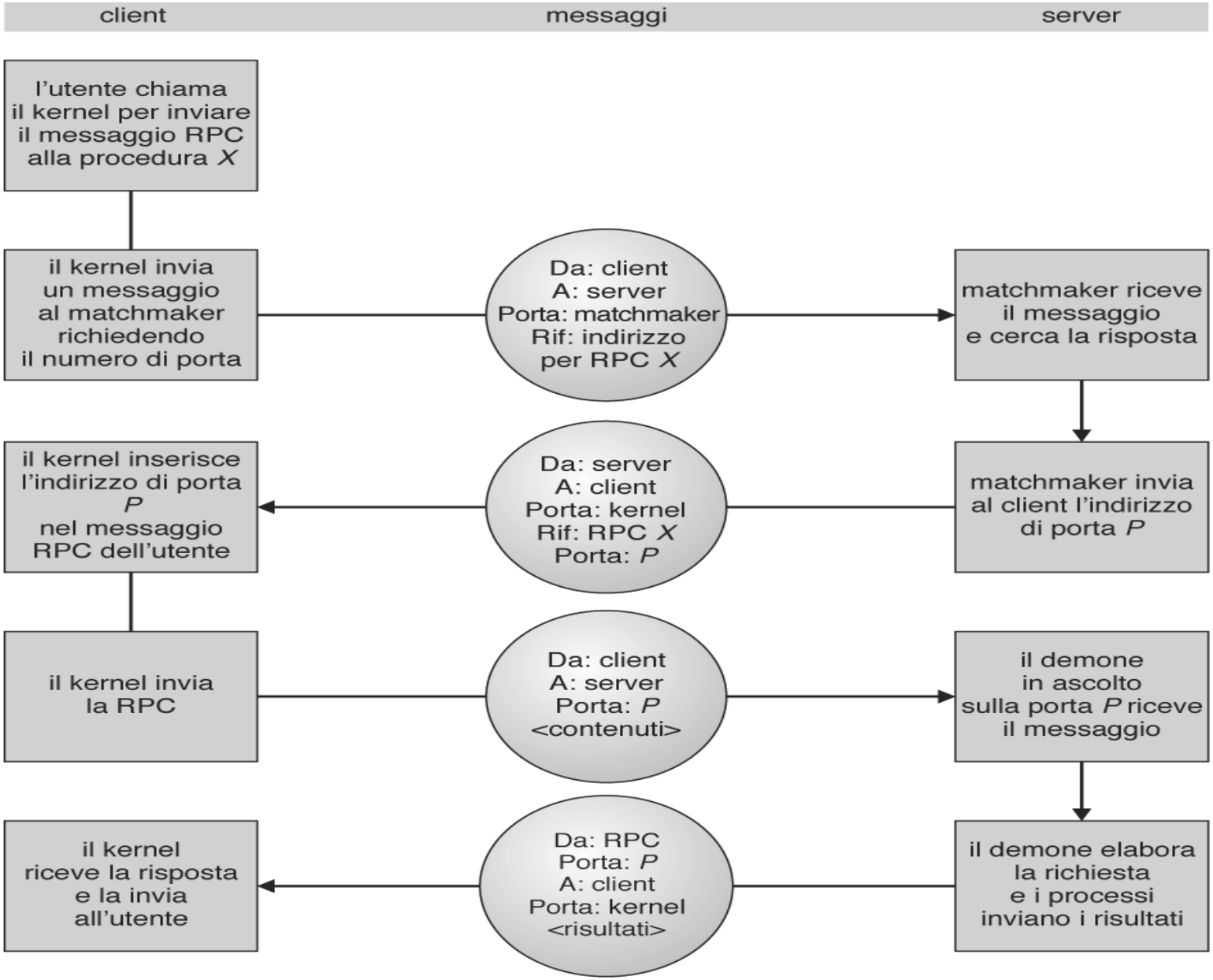
- La **semantica (b) "esattamente una volta"** garantisce l'esecuzione della chiamata
  - Il server implementa il protocollo (a) e in più notifica al client che la chiamata RPC è stata ricevuta ed eseguita (msg ACK di acknowledgement)
  - Il client deve rispedire ciascuna chiamata RPC periodicamente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

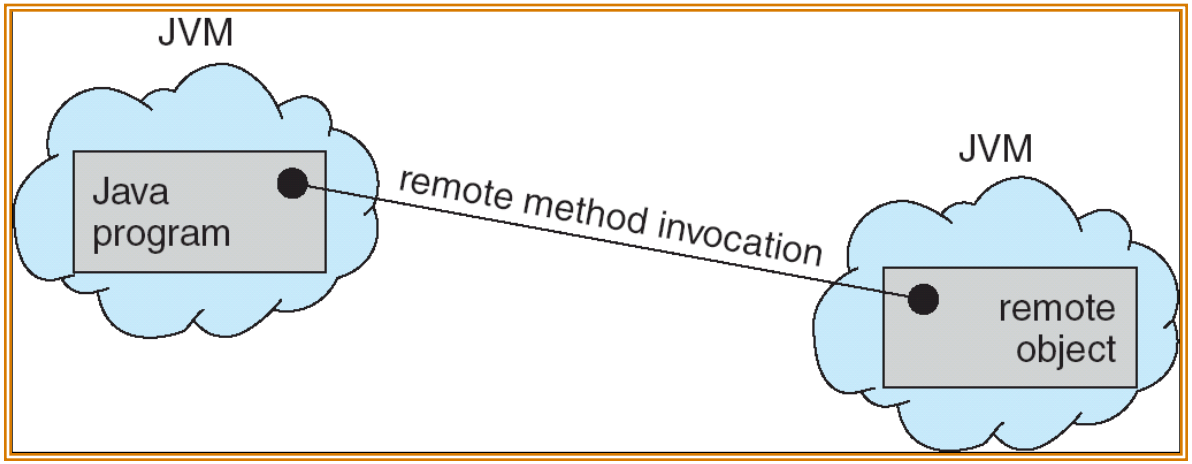
1. Associazione fissa e nota RPC-Porta
2. Dinamicamente tramite un servizio di Rendezvous
   - Il server ha un demone in ascolto che riceve una richiesta dal client e restituisce il numero di porta (slide successiva)

Esecuzione di una RPC con demone rendezvous

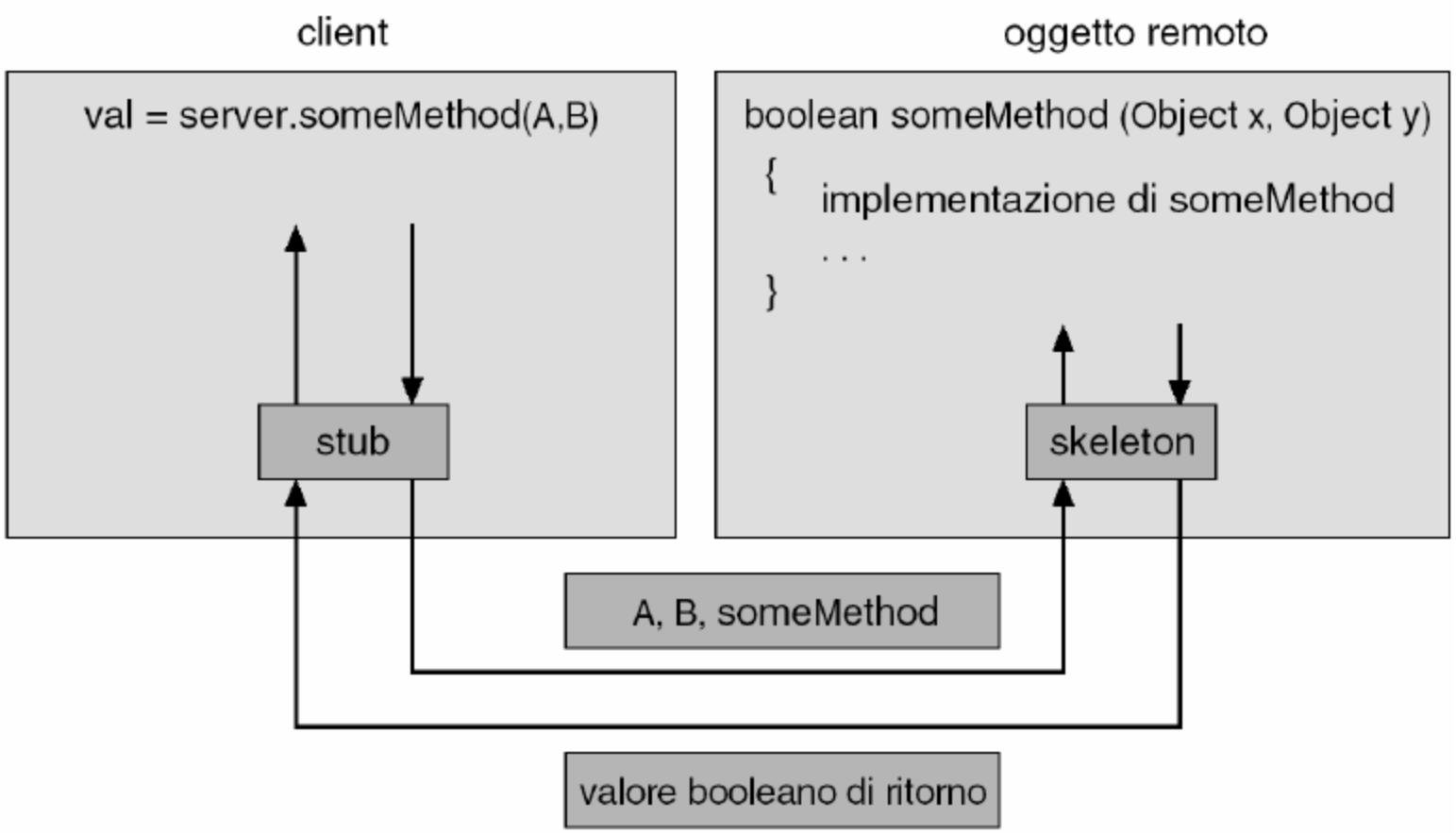


Remote Method Invocation (RMI)

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

Esempio RMI

L'interfaccia remoteDate dichiara un metodo che restituisce una data

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

Esempio RMI – Lato server

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

```
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);
      }
   }
}
```

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)

Esempio RMI – Lato Client

```
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);
      }
    }
}
```