Comunicazione tra Processi

Programmi come Insiemi di Processi/Thread

- I sistemi di elaborazione distribuiti e gli algoritmi distribuiti si basano sulla esecuzione contemporanea di programmi residenti su più calcolatori.

- L'uso dei thread ha permesso di migliorare le prestazioni e le funzionalità dei sistemi distribuiti e delle applicazioni che fanno uso di insieme di elaboratori connessi tra loro.

- Per questo è essenziale disporre di meccanismi per la creazione e attivazione di **processi** e **thread** e di meccanismi per la loro comunicazione e sincronizzazione.

Comunicazioni in un Sistema Distribuito

- Un sistema software distribuito è realizzato tramite un insieme di processi che a volte hanno bisogno di comunicare, di sincronizzarsi, e di cooperare.

- Il meccanismo di comunicazione di più basso livello in un sistema distribuito è lo **scambio di messaggi**.

- Su di esso possono essere costruiti meccanismi di comunicazione più semplici da usare:
  - Remote procedure call,
  - Active messages,
  - Publish/subscribe,
  - Streams

Comunicazione tra Processi/Thread

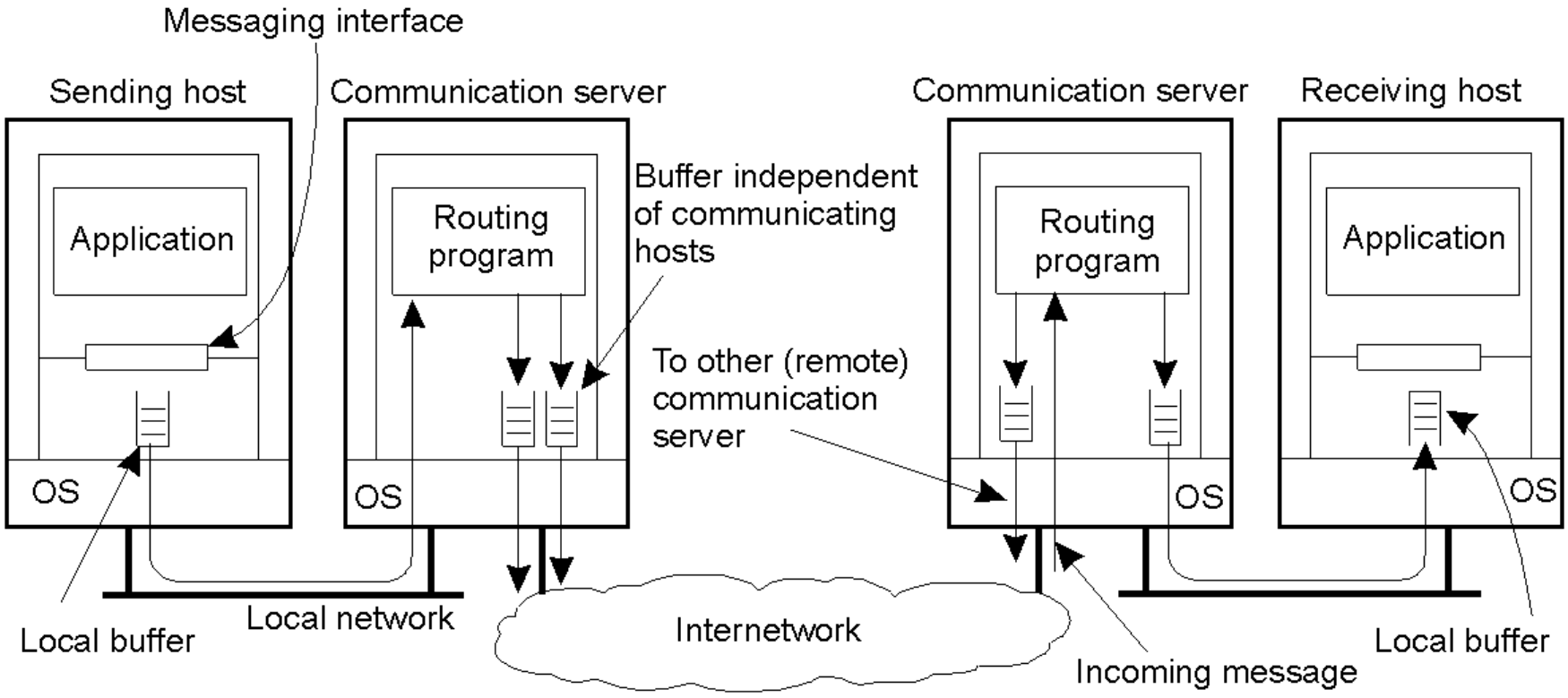
- Le comunicazioni possono essere anche
  - **Sincrone**: il mittente si blocca fino a che il destinatario riceve il messaggio

    o
  - **Asincrone**: il mittente continua senza attendere che il destinatario riceva il messaggio.

- Queste si possono combinare con le comunicazioni
  - **Persistenti**: il messaggio viene conservato fino a che verrà consegnato

    o
  - **Transienti**: il messaggio viene consegnato soltanto se mittente e destinatario sono in esecuzione contemporaneamente.

Comunicazione basata su Messaggi



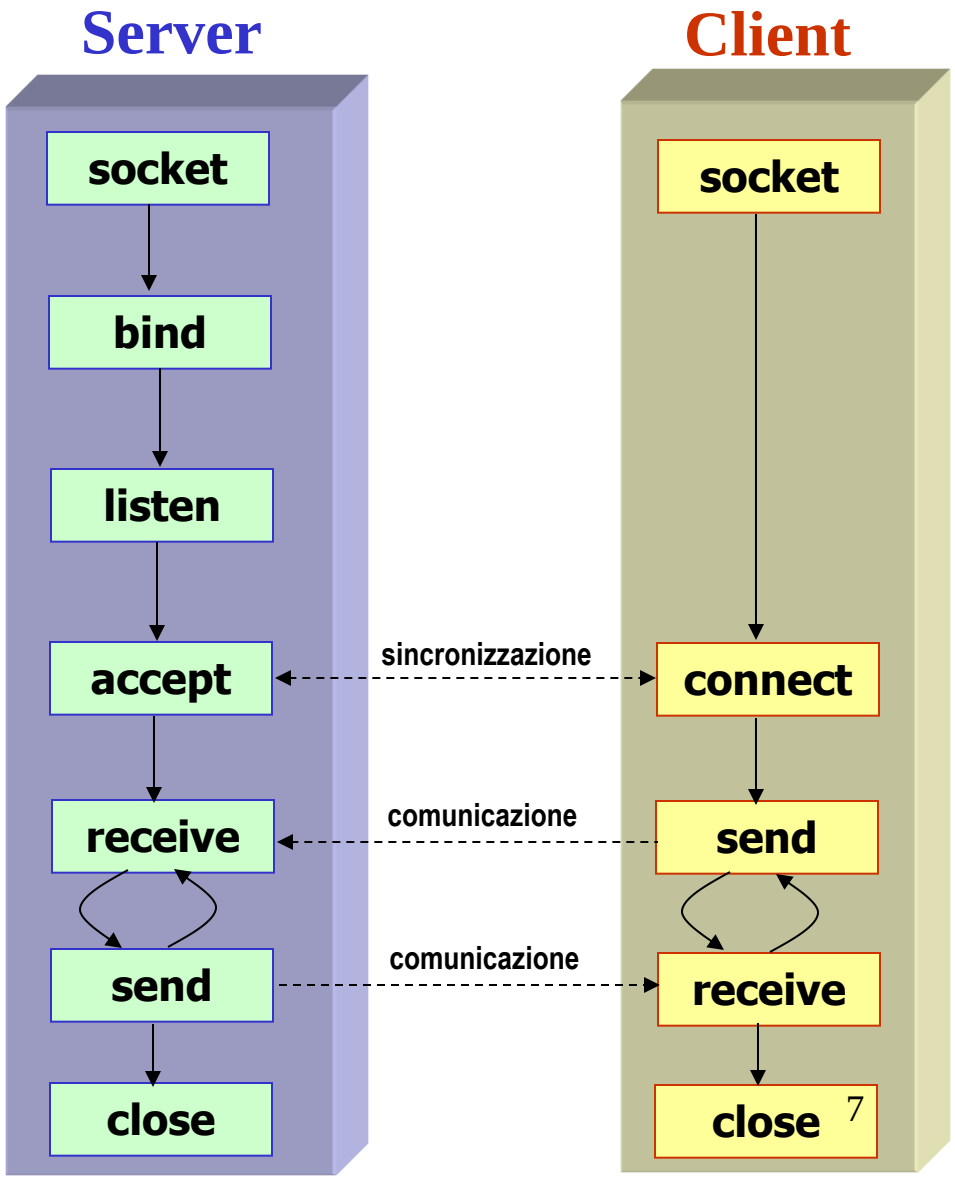
Organizzazione generale di un sistema di comunicazione in cui i nodi sono connessi in rete.

Comunicazione basata su Messaggi: Socket

- **Socket** : porte di comunicazione software per lo scambio di messaggi tra processi.
- Interfacce di comunicazione UDP (senza connessione) e TCP (con connessione).
- Interfaccie simili per comunicazioni tra processi locali (sullo stesso computer) o tra processi remoti (su computer diversi connessi in rete): AF_UNIX e AF_INET.
- Implementate inizialmente in Unix Berkley 4.3, disponibili su molti sistemi operativi. Su Windows: WinSock

Socket TCP (con connessione)

- Primitive:
  - `socket`
  - `bind`
  - `listen`
  - `accept`
  - `connect`
  - `send/write`
  - `receive/read`
  - `close`
  - `select`



Client Socket TCP (con connessione)

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include …

int main(int argc, char *argv[])
{int sockfd = 0, n = 0; char recvBuff[1024]; struct sockaddr_in serv_addr;
 if(argc != 2)
   {printf("\n Usage: %s <ip of server> \n",argv[0]); return 1;}
 memset(recvBuff, '0',sizeof(recvBuff));
 if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {printf("\n Error : Could not create socket \n"); return 1;}
 memset(&serv_addr, '0', sizeof(serv_addr));
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_port = htons(5000);
 if(inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0)
   {printf("\n inet_pton error occured\n"); return 1;}
 if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {printf("\n Error : Connect Failed \n"); return 1;}
 while ((n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0)
   {recvBuff[n] = 0;
    if(fputs(recvBuff, stdout) == EOF)
      {printf("\n Error : Fputs error\n"); } }
    if(n < 0) { printf("\n Read error \n"); } return 0; close(sockfd);
}
```

Server Socket TCP (con connessione)

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include …

int main(int argc, char *argv[])
{int listenfd = 0, connfd = 0; struct sockaddr_in serv_addr;
  char sendBuff[1025]; time_t ticks;
  listenfd = socket(AF_INET, SOCK_STREAM, 0);
   memset(&serv_addr, '0', sizeof(serv_addr));
   memset(sendBuff, '0', sizeof(sendBuff));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_addr.sin_port = htons(5000);
   bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
   listen(listenfd, 10);
   while(1)
   { connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
     ticks = time(NULL);
     snprintf(sendBuff, sizeof(sendBuff),  "%.24s\r\n", ctime(&ticks));
     write(connfd, sendBuff, strlen(sendBuff));
     close(connfd);
     sleep(1);
   }
  }
```

Select Socket

```
int select( int nfds, fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

- Permette il controllo di comunicazioni su più socket in lettura, scrittura e errore per un certo intervallo di tempo.
- Ritorna il numero di socket pronti per la comunicazione e permette di selezionare quello con il quale comunicare.
- Senza l'operazione di select non si può realizzare l'attesa non deterministica.

Java NIO Selector

- Usare il package `java.nio`
- Creare un *Selector*

```
Selector selector = Selector.open();
```

- Registrare i diversi canali *socket*

```
channel.configureBlocking(false);
SelectionKey K = channel.register(selector, SelectionKey.OP_READ);
```

- Selezionare i socket «pronti»

```
int channels = selector.select();
```

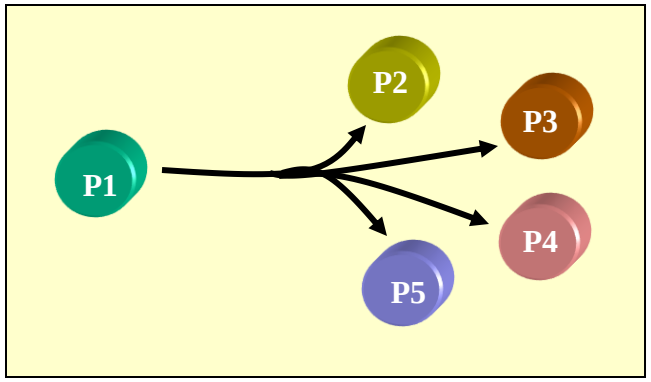
- Estrarre gli id (chiavi) dei socket «pronti»

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

Comunicazione Multicast

- Lo scambio di messaggi non necessariamente deve avvenire tra due processi (punto a punto).

- La comunicazione **multicast** permette l'invio di un messaggio da 1 processo ad un gruppo di N processi scelti secondo una qualche regola o in base ad una lista.




```
P1::

send(P2,P3,P4,P5; M);
```

- I socket permettono la comunicazione multicast.

Comunicazione Broadcast

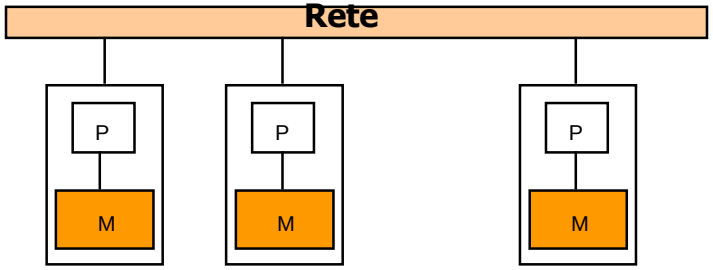
- Se l'insieme dei processi a cui si invia un messaggio è costituito da **tutti i processi** che compongono una applicazione distribuita, la comunicazione si dice **broadcast** (diffusione).

- Questo modello di comunicazione è utile in applicazioni in cui occorre distribuire le informazioni.

- Puo' limitare la scalabilità delle applicazioni composte da un elevato numero di thread/processi.

Linguaggi a Memoria Distribuita

- Questi linguaggi riflettono il modello dei calcolatori a memoria distribuita composti da un insieme di elementi di elaborazione connessi da una rete (multicomputers, clusters, LAN).



- In questo modello un programma parallelo consiste da un insieme di processi in esecuzione su più processori che cooperano tramite lo scambio di messaggi (message passing).

- Due principali aspetti in questo tipo di programmazione sono la creazione/attivazione dei processi concorrenti ed la loro cooperazione.

Linguaggi a Memoria Distribuita

- Alcuni forniscono delle primitive per la creazione esplicita dei processi durante l'esecuzione del programma (*creazione dinamica*):

  `fork/join`, **`new`** e **`create`**.

- In altri il numero dei processi è definito a tempo di compilazione (*creazione statica*):

  `par`, **`parbegin`**, **`cobegin/coend`**.

MPI

- MPI (*Message Passing Interface*) è una libreria standard per lo sviluppo di programmi paralleli e distribuiti attraverso primitive di scambio messaggi (alcuine centinaia).

- MPI è disponibile per macchine massicciamente parallele, reti di workstation eterogenee, PC, etc. → applicazioni portabili

- Un programma parallelo in MPI è strutturato come una collezione di processi concorrenti che eseguono programmi scritti in un linguaggio sequenziale con chiamate ad una libreria (MPI) per realizzare lo scambio di messaggi.

- In MPI-1 non c'è creazione di processi ma è disponibile in MPI-2 e in MPI-3.

MPI

- La libreria MPI contiene funzioni per supportare la comunicazione punto-a-punto fra coppie di processi, come ad esempio

```
MPI_Send(mess, strlen(mess)+1, type, 1, tag, MPI_COM);
MPI_Recv(mess, leng, type,0, tag, MPI_COM, &status);
```

- Le funzioni per comunicazioni collettive all'interno di gruppi di processi come:

```
MPI_Bcast (inbuf, incnt, intype, root, comm);
MPI_Gather (outbuf, outcnt, outype, inbuf, incnt,..);
```

- MPI offre un modello di programmazione di basso livello, tuttavia anch'esso è molto usato molto a causa della sua portabilità.

MPI

- Alcune principali operazioni di MPI per gestire comunicazioni transienti:

Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

MPI

- Per eseguire un programma in MPI

  `mpirun – np <N>` `program`

  `<N>` indica il numero di processi che verranno eseguiti.

- Si usano **`MPI_Init`** e **`MPI_Finalize`** per creare e terminare l'ambiente MPI e **`MPI_Comm_rank e MPI_Comm_size`** per avere l'id del processo e il numero di processi dell'applicazione.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

MPI

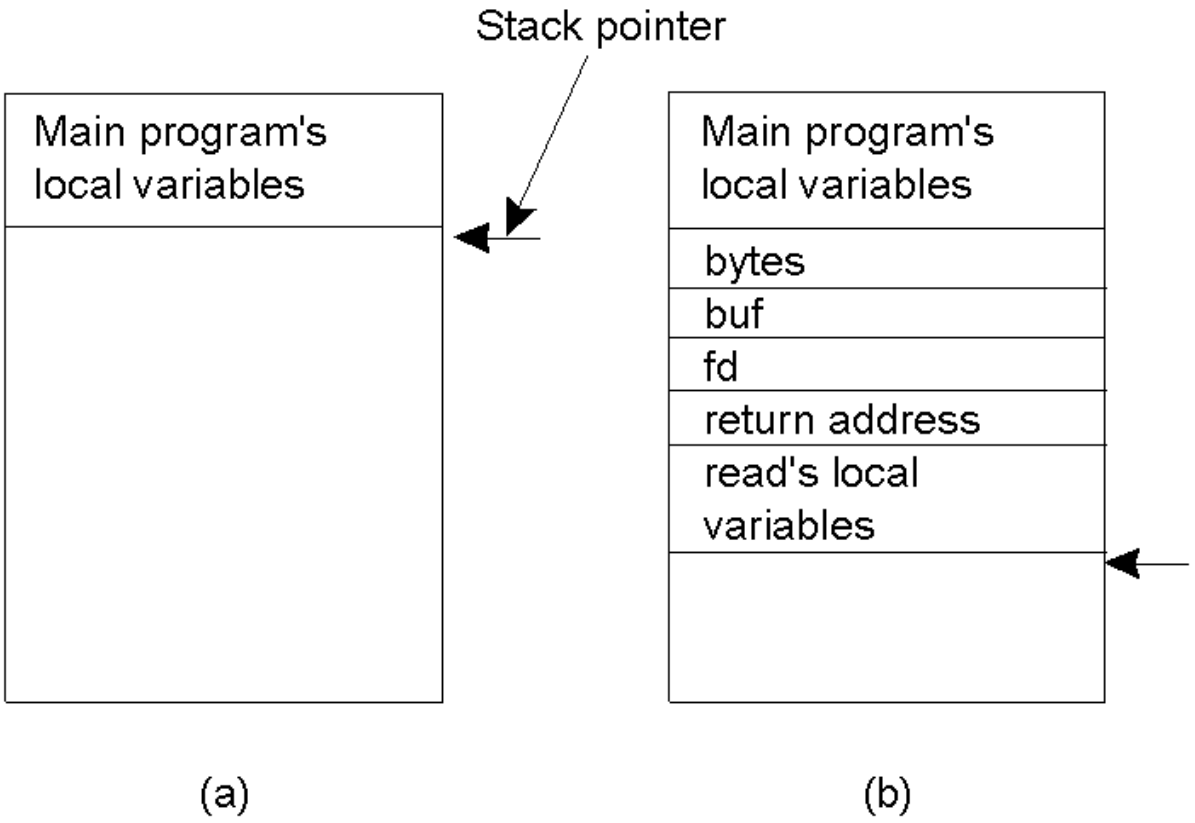
```
// Programma send_receive per lo scambio di messaggi tra 2 processi;

#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int num, int w_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &w_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &w_size);
  if (w_rank == 0)
  {
   num = 15;
   MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (w_rank == 1)
  {
   MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   printf("Process 1 received number %d from process 0\n", num);
  }
  MPI_Finalize();
}
```

mpirun –np 2 ./send_recv

Chiamata di Procedura Remota

- La Chiamata di procedura remota è molto simile ad una chiamata di procedura/funzione/metodo tradizionale.
- La principale differenza è che la procedura viene eseguita da un processo/thread differente che può essere in esecuzione sullo stesso computer o su un computer remoto connesso alla rete del processo/thread che ha invocato la procedura.
- Il client deve invocare la procedura correttamente e il server deve contenere il codice eseguibile della procedura.
- Esempi: RPC di Unix, RMI di Java.

Chiamata di Procedura Convenzionale

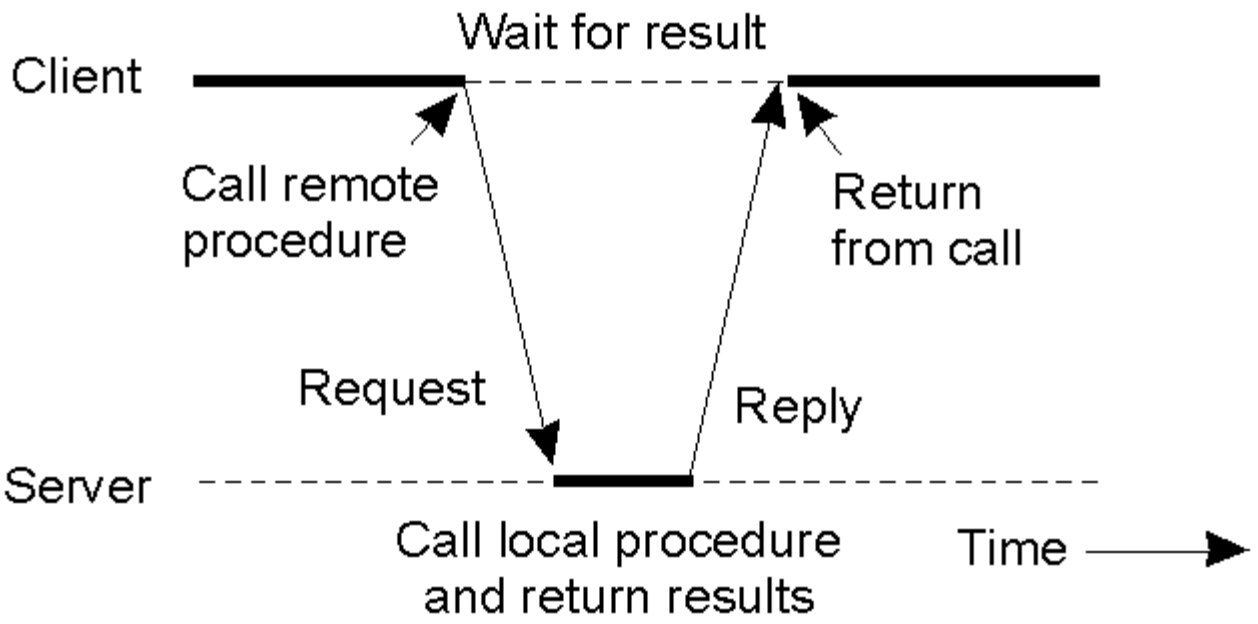


Passaggio di parametri in una chiamata di procedura (read):

a) lo stack prima della chiamata

b) lo stack mentre la chiamata della procedura è attiva

Stubs per Client e Server

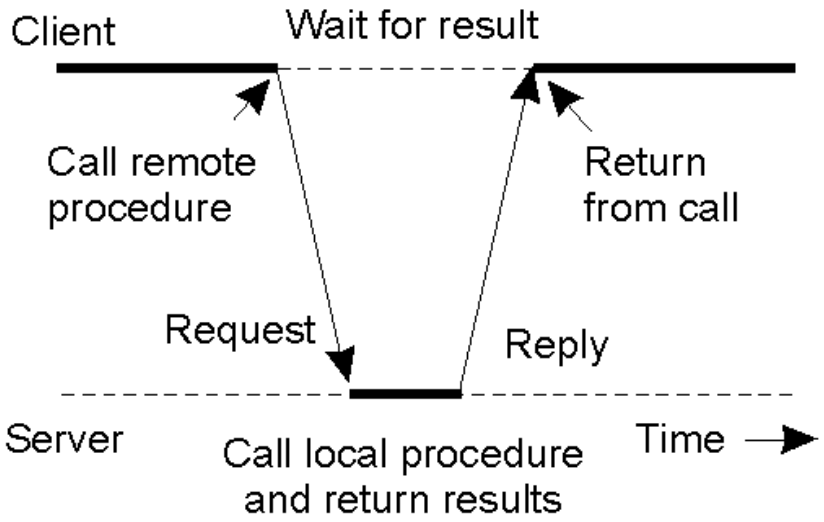


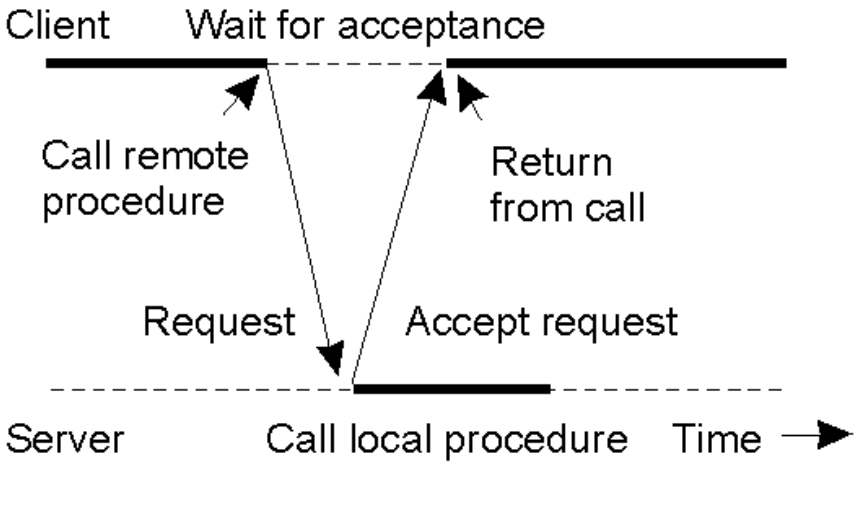
Schema di una RPC tra un programma cliente and programma server.

Passi di una Remote Procedure Call

1. La chiamata di procedura del Client chiama un Client stub (routine di gestione dei paramentri che gestisce il ***marshaling***)
2. Il Client stub costruisce un messaggio e lo passa al SO locale
3. Il SO locale invia un messaggio al SO remoto
4. Il SO remoto passa il messaggio al Server stub
5. Il Server stub preleva i parametri e invoca il Server
6. Il Server effettua le operazioni, ritorna il risultato al Server stub
7. Il Server stub mette il risultato in un messaggio e chiama il SO del server
8. SO del server invia il messaggio al SO del client
9. Il SO del client passa il messaggio allo stub del client
10. Il Clent stub preleva il risultato e lo ritorna al Client.

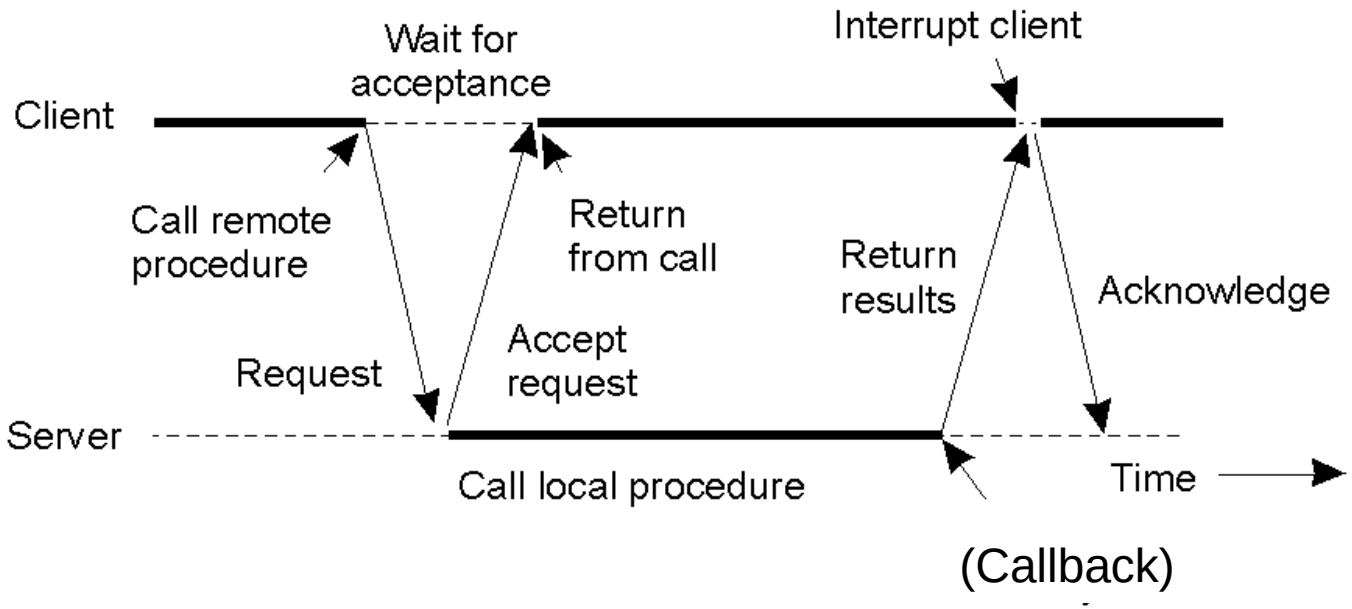
RPC Asincrona (1)





a) L'interazione tra client e server in una RPC tradizionale
b) L'interazione usando una RPC asincrona

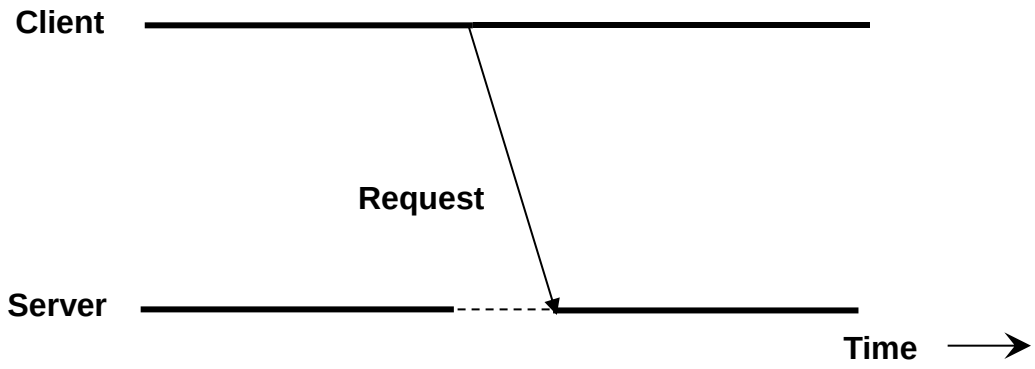
RPC Asincrona (2)



Un client e un server interagiscono con RPC asincrone

RPC Asincrona (3)

- RPC asincrone One-way
  - Il client continua dopo avere effettuato una chiamata di procedura remota
  - Simile ad una **send** senza risposta
  - Affidabilità non assicurata: il cliente non sa se la richiesta verrà servita.



RPC Multicast

- RPC multiple contemporanee
  - Usa più RPC One-way per inviare le chiamate a un gruppo di server.
  - I risultati sono riportati al client tramite delle Callback
  - Il client può non conoscere il numero esatto dei server e potrebbe attendere tutte le risposte o soltanto alcune.