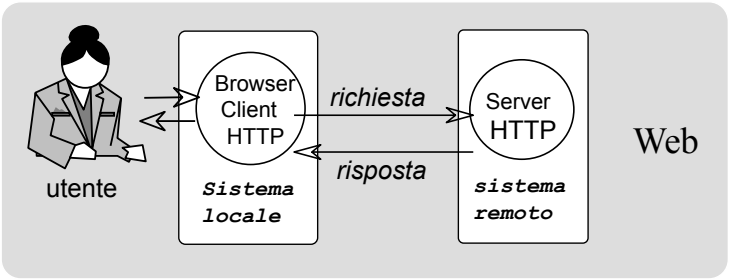
La comunicazione tra processi in una rete di calcolatori



*Un'**applicazione distribuita** è costituita da processi distinti per località che comunicano e cooperano attraverso lo scambio di messaggi, per ottenere risultati coordinati.*

Quali sono gli aspetti principali per la realizzazione di un'applicazione distribuita?

- Identificazione dei processi (nomi)
- Primitive di comunicazione tra processi
- Sincronizzazione dei processi
- Comunicazioni con/senza connessione
- Affidabilità
- Eterogeneità e Formato dei dati
- Modelli di interazione (Client/Server e altri modelli)

Identificazione dei processi comunicanti (sistema di nomi)

Necessità di definire un sistema di identificazione per le diverse entità.

Il primo problema da affrontare riguarda l'**identificazione dei processi comunicanti** nella rete.

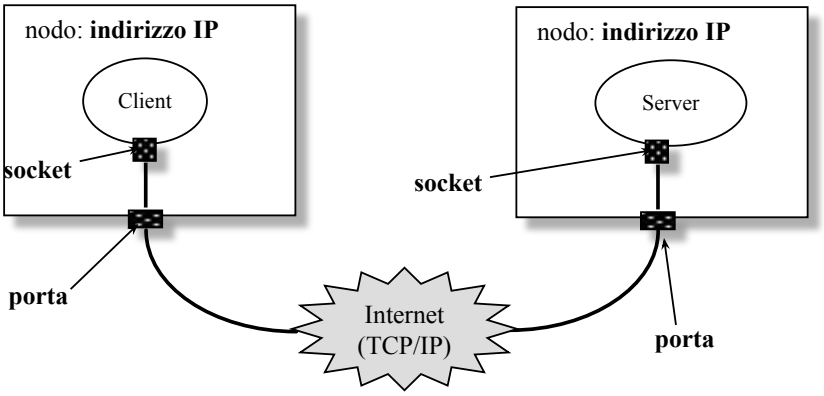
Per ogni processo bisogna definire un **nome globale** univoco e sempre non ambiguo, per esempio, tipicamente:

«nome» della macchina + «nome» del processo all'interno della macchina

Questo problema è risolto dai protocolli di comunicazione sottostanti, che nel caso di Internet sono i protocolli di trasporto TCP/UDP e di rete IP.

Nome macchina = indirizzo IP (o nome logico)
Nome processo = come identificare un processo? Il process identifier (pid) è inadeguato, uso di «porte».

L'identificazione dei processi comunicanti in Internet



Una macchina è identificata univocamente da un **indirizzo IP** (4 byte, es. 192.167.215.12). La **porta** è un numero intero di 16 bit (astrazione fornita da **TCP** e **UDP**) che rappresenta un **identificativo univoco di servizio**, che rende possibile identificare un processo senza dover conoscere il suo pid.

I messaggi sono consegnati su una specifica porta di una macchina, non direttamente a un processo. Un processo si **lega** a una porta per ricevere (o spedire) dei messaggi (es. di porte, 80 per i server Web, 25 per la posta, etc.).

Un ***indirizzo IP*** e una ***porta*** rappresentano un endpoint di un canale di comunicazione.

Primitive di comunicazione

Molte primitive e modelli di comunicazione con scelte diverse su due dimensioni:

a) designazione dei processi sorgente e destinatario della comunicazione:

- schemi **diretti simmetrici**
- schemi **diretti asimmetrici**
- schemi **indiretti**

b) tipo di sincronizzazione tra i processi comunicanti:
comunicazione **sincrona** e **asincrona**

Caratteristiche a) e b) ortogonali: le soluzioni sono tra loro indipendenti

Uso di primitive send/receive:

```
send(messaggio) to Pdest        [send(msg, P)]
receive(&messaggio) from Psorg   [receive(msg, Q)]
```

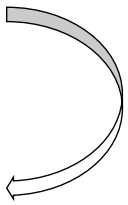
Designazione di sorgente e destinatario
schemi diretti simmetrici

In tale tipo di schema, i processi si nominano **esplicitamente:**

```
send(messaggio) to Pdest
receive(&messaggio) from Psorg
```

È utilizzato nei modelli di tipo pipeline (es. Unix)

```
. . . .
pipe(p);
if ((pid=fork())==0) { /*FIGLIO*/
    close(p[0]);
    num=servizio();
    write(p[1],&num, dim);
    exit(0);
}
close(p[1]);
read(p[0],&ris, dim);
. . . .
```



Designazione di sorgente e destinatario
schemi diretti asimmetrici

In tale schema, il mittente **nomina esplicitamente** il destinatario, ma questi al contrario non esprime il nome del processo da cui desidera ricevere messaggi.

Schema **molti a uno** (modello **Cliente/Servitore**). I processi cliente specificano il destinatario (servitore) delle loro richieste. Il processo servitore è pronto a ricevere messaggi da qualunque cliente.

Attenzione! La receive nel server è *asimmetrica*, mentre nel client è *simmetrica*.

```
Pi(Cliente)   Pj(Servitore)
......
send (msg) to Pj receive (&request, &Pi)
   <esecuzione del servizio>
receive (&ris) from Pj  send (response) to Pi
```

Semantica corrispondente all'uso di un processo come gestore di una risorsa.

Gli schemi asimmetrici sono utili per comunicazioni sia da **uno a molti** che da **molti a molti**: ad esempio N processi client che inviano richieste a un qualunque servitore, scelto tra un insieme di M servitori equivalenti.

Designazione di sorgente e destinatario
schemi indiretti

```
send(msg, M)

receive(msg, M)
```

La comunicazione avviene **tramite un oggetto mailbox `M`**.

Il supporto mette a disposizione delle chiamate di sistema per

- creare una mailbox
- inviare e ricevere messaggi alla/dalla mailbox
- distruggere la mailbox

Tipi di sincronizzazione tra processi
Modalità Sincrona e Asincrona

In un'applicazione distribuita, ogni processo esegue senza conoscere lo stato di esecuzione degli altri. Quando due processi comunicano, sorge un problema di sincronizzazione. Per esempio, un browser visualizza una pagina Web che è il frutto di molti processi operanti sul lato Server.

La sincronizzazione è legata all'uso di primitive Sincrone o Asincrone.

Una primitiva sincrona BLOCCA il processo fino al termine della sua esecuzione.

Il blocco e lo sblocco delle primitive è ovviamente realizzato e gestito dal supporto alla comunicazione senza intervento del programmatore, che deve però selezionare le primitive con la modalità più opportuna per l'applicazione.

Sincronizzazione tra processi - Modalità Sincrona

Nel caso di modalità sincrona, il processo si **blocca** in attesa del completamento dell'operazione richiesta:

- **send sincrona**: processo mittente si blocca fino a che il messaggio viene ricevuto dal destinatario.
- **receive sincrona**: processo destinatario si blocca fino alla ricezione del messaggio.

Quindi, in questo caso, un messaggio ricevuto contiene informazioni corrispondenti allo **stato attuale** dell'altro processo.

Possibili problemi di **deadlock** (blocco critico) in caso di errori di programmazione nell'uso di send/receive o perdita messaggi. Uso di timeout o di multithreading.

Si noti che la modalità sincrona è **molto onerosa** e si presta più per soluzioni di alto livello (es. RPC) che per lo scambio di messaggi.

Sincronizzazione tra processi - Modalità Asincrona

Nel caso di modalità asincrona, il processo **continua la sua esecuzione** immediatamente dopo che l'operazione è stata invocata:

- **send asincrona**: processo mittente continua esecuzione subito dopo la send, senza sapere se il destinatario ha ricevuto il messaggio.
- **receive asincrona**: processo destinatario esegue la receive senza bloccarsi ma nel buffer di ricezione non è detto che ci sia un messaggio. Se il messaggio non è arrivato, due casi: *polling* (si invoca continuamente la receive) e *gestione a eventi* (il supporto di IPC notifica al processo ricevitore la presenza di messaggio da ricevere).

Non c'è garanzia che il messaggio sia stato ricevuto.

Il messaggio ricevuto non contiene informazioni che possano essere associate allo **stato attuale** del mittente (difficoltà di verifica dei programmi).

Tipi di sincronizzazione tra processi
Modalità Sincrona e Asincrona

Tipi di send:
- send sincrona (bloccante)
- send asincrona (non bloccante)

Tipi di receive:
- receive sincrona (bloccante)
- receive asincrona (non bloccante)

Si possono avere tutte le combinazioni di send e receive.

In generale, all'interno di un'applicazione si tende a utilizzare solo pochi tipi di modalità di sincronizzazione, senza mescolare differenti comportamenti.

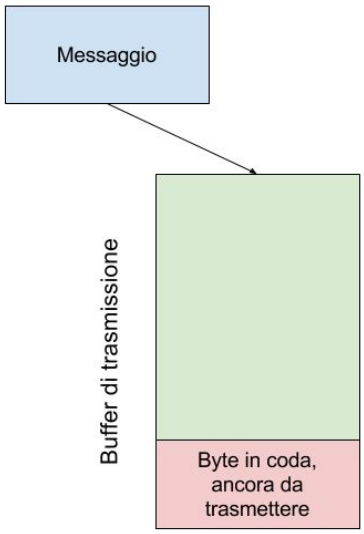
Sincronizzazione e Buffering

Per ragioni di performance, tipicamente si effettua *il buffering dei messaggi* che *porta a una semantica di sincronizzazione leggermente diversa per le primitive send e receive*.

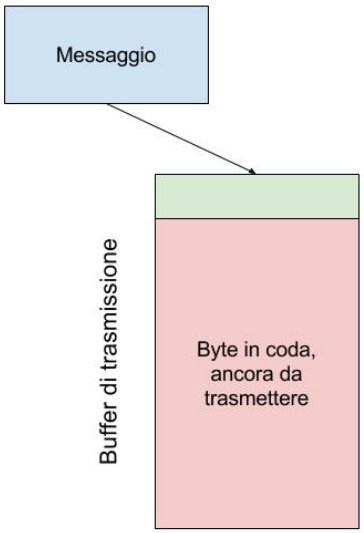
Più precisamente, prima della trasmissione (nel caso di send) o della consegna all'applicazione dei dati letti (nel caso di receive) i messaggi scambiati attraverso primitive IPC risiedono in una memoria temporanea all'interno del kernel. Questo consente di accorpare le operazioni di I/O (trasmissione di messaggi in rete, copia di dati tra kernel space e user space), con notevole impatto sulla performance dell'applicazione.

Si noti che **il buffering comporta un tradeoff**: a migliori performance corrisponde un minore controllo. Infatti, il mittente non è mai a conoscenza dello stato del messaggio che invia tramite send. Può pertanto essere necessaria l'adozione di meccanismi che verifichino la corretta consegna dei messaggi a destinazione.

Sincronizzazione e Buffering: caso send

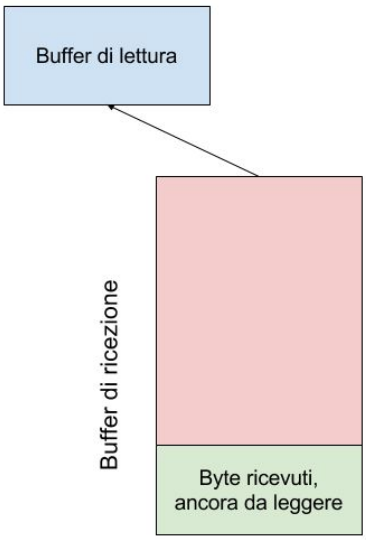


Nel caso vi sia spazio sufficiente nel buffer di trasmissione, il kernel prende in carico il messaggio da inviare e send ritorna immediatamente.

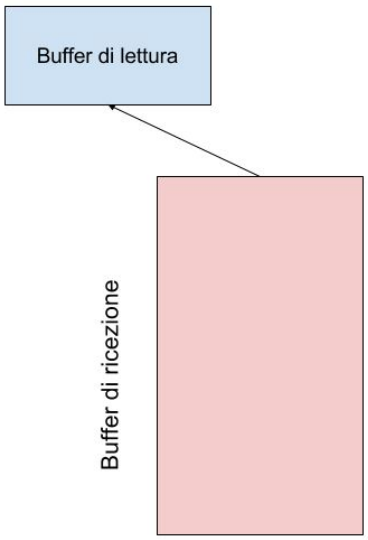


Nel caso non vi sia spazio sufficiente nel buffer di trasmissione, il kernel blocca la send finché non sarà in grado riuscire a prendere in carico il messaggio.

Sincronizzazione e Buffering: caso receive



Nel caso vi siano dati nel buffer di ricezione, il kernel li copia nel buffer di lettura passatogli da receive, e quest'ultima funzione ritorna immediatamente.



Nel caso il buffer di ricezione sia vuoto, il kernel blocca la receive finché non avrà ricevuto dei dati da copiare nel buffer di lettura.

Comunicazione con/senza connessione

Applicazioni distribuite fanno uso di canali di comunicazione con connessione o senza connessione:

- **Con Connessione**. Viene stabilita una connessione tra i processi chiamante e chiamato. Per esempio il sistema telefonico.
  La connessione può essere logica oppure fisica.
  Stabilita la connessione, i dati possono fluire continuamente tra i processi.

- **Senza Connessione.** Non c'è connessione tra i processi. Ogni messaggio deve essere singolarmente indirizzato e consegnato al processo destinatario. Per esempio, il sistema postale.
  Diversi messaggi spediti da un processo a un altro possono seguire strade diverse e anche arrivare non nell'ordine di spedizione.

Affidabilità della comunicazione

Il supporto di Inter Process Communication utilizzato può fornire diverse garanzie nella consegna dei messaggi:

- **Comunicazione Affidabile**. Nel caso in cui il supporto garantisca la consegna dei messaggi. In caso di perdita di messaggi o guasti, il supporto tipicamente ritrasmette (anche più volte) il messaggio.
  Ovviamente non può risolvere guasti permanenti (per esempio di rete).
  Alto costo di realizzazione (basse prestazioni) Vs. semplicità di programmazione.

- **Comunicazione Non Affidabile.** In questo caso il supporto invia i messaggi senza verificarne la consegna.
  Alte prestazioni Vs. difficoltà di programmazione.

Dalla teoria alla pratica…

Nonostante in teoria sia possibile combinare le caratteristiche delle IPC viste finora in numerosi modi, in pratica si usano (quasi) esclusivamente 2 modalità:

	Socket stream	**Socket datagram**
Schema comunicazione client	Diretto simmetrico	Diretto simmetrico
Schema comunicazione server	Diretto asimmetrico	Diretto asimmetrico
Modalità sincronizzazione send	Sincrona, con buffering	Sincrona, con buffering
Modalità sincronizzazione receive	Sincrona, con buffering	Sincrona, con buffering
Comunicazione con connessione	Sì	No
Comunicazione affidabile	Sì	No
Protocollo di comunicazione sottostante	TCP	UDP