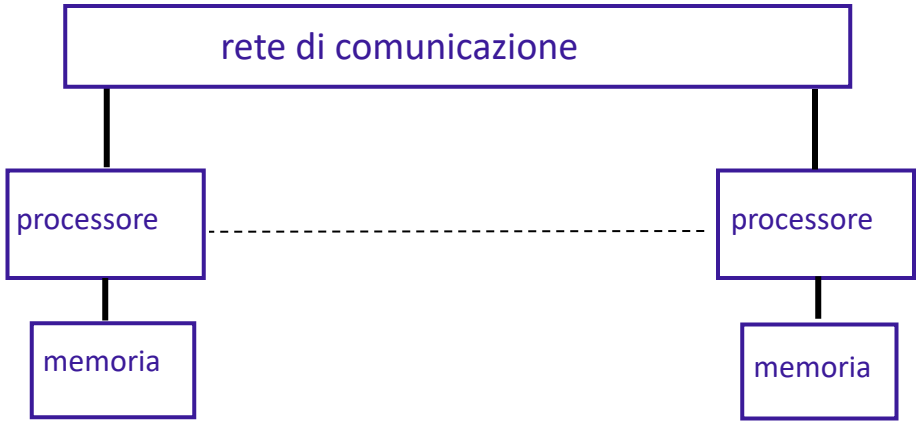
Il Modello a scambio di messaggi

Caratteristiche del modello

modello architetturale di macchina (virtuale) concorrente



Caratteristiche del modello

- Ogni processo può accedere **esclusivamente** alle risorse allocate nella **propria memoria locale**.

- Ogni risorsa del sistema è accessibile direttamente ad **un solo processo (il Gestore della risorsa)**

- Se una risorsa è necessaria a più processi applicativi, ciascuno di questi **(processi clienti**) dovrà delegare l'unico processo che può operare sulla risorsa (processo **gestore**, o server) all'esecuzione delle operazioni richieste; al termine di ogni operazione il gestore restituirà al cliente gli eventuali risultati.

Caratteristiche del modello

In questo modello il concetto di **gestore** di una risorsa coincide con quello di processo server.

Ogni processo, per usufruire dei servizi offerti da una risorsa, dovrà **comunicare** con il gestore

- il meccanismo di base utilizzato dai processi per qualunque tipo di interazione è costituito dal **meccanismo di scambio di messaggi**.

Canali di comunicazione

canale = collegamento logico mediante il quale due o più processi comunicano.

Il **nucleo** della macchina concorrente realizza l'astrazione “canale” come meccanismo primitivo per lo scambio di informazioni.

È compito del linguaggio di programmazione offrire gli **strumenti linguistici** di alto livello per:

- **specificare i canali** di comunicazione
- **utilizzarli per esprimere le interazioni** tra i processi

Caratteristiche del canale di comunicazione

I **parametri** che caratterizzano il concetto di canale sono:

1. la **direzione del flusso dei dati** che un canale può trasferire;

2. la **designazione** del **canale** e dei processi **origine** e **destinatario** di ogni comunicazione;

3. il tipo di **sincronizzazione** fra i processi comunicanti.

Tipi di canale

Facendo riferimento alla **direzione del flusso dei dati**, possiamo distinguere 2 tipi di canale:

1. Canale **monodirezionale**: consente il flusso di messaggi in una sola direzione (mittente verso ricevente).

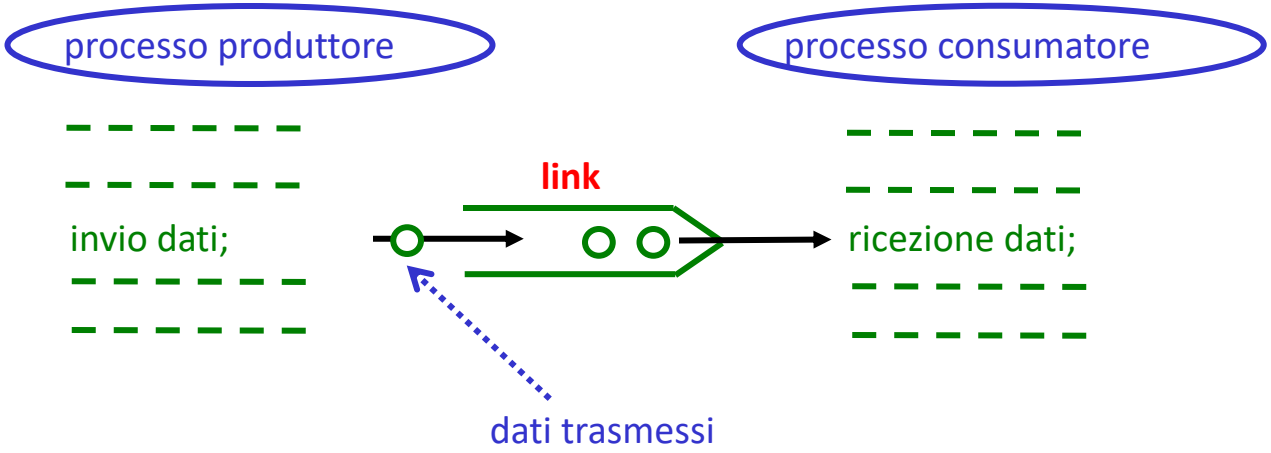
2. Canale **bidirezionale**: può essere usato sia per inviare che per ricevere informazioni.

Tipi di canale

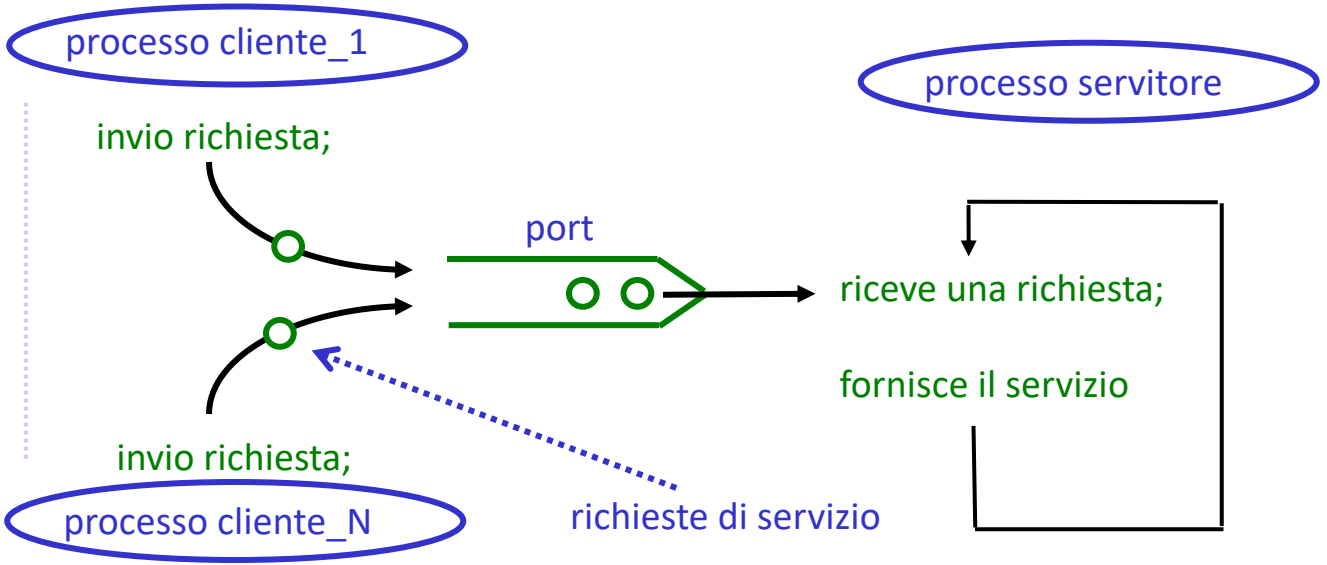
Facendo riferimento alla **designazione dei processi** comunicanti, è possibile definire tre tipi di canale:

1. **link**: da-uno-a-uno (canale simmetrico)
2. **port**: da-molti-a-uno (canale asimmetrico)
3. **mailbox**: da-molti-a-molti (canale asimmetrico)

Link: comunicazione simmetrica produttore-consumatore



Port: comunicazione asimmetrica: es. cliente-servitore



Tipi di canale

Con riferimento alla modalità di **sincronizzazione** tra i processi comunicanti, possiamo individuare **3 tipi** di canale:

a) Comunicazione **asincrona**

b) Comunicazione **sincrona**

c) Comunicazione con **sincronizzazione estesa**

Comunicazione asincrona

Semantica:

Il processo mittente **continua la sua esecuzione** immediatamente dopo l'invio del messaggio.

Effetti:

- La ricezione del messaggio può avvenire in un istante successivo all'invio: il messaggio ricevuto contiene informazioni che **non possono essere attribuite allo stato attuale** del mittente => difficoltà nella verifica dei programmi.

- L'invio di un messaggio **non è un punto di sincronizzazione per mittente e destinatario**.

Comunicazione asincrona

Proprietà:

- **Carenza espressiva** (difficoltà di verifica dei programmi)
- L'**assenza di vincoli di sincronizzazione** tra mittente e destinatario favorisce il **grado di concorrenza** .
- Da un punto di vista realizzativo, sarebbe necessario un **buffer di capacità illimitata**.

Comunicazione asincrona

Realizzazione:

Ogni implementazione prevede inevitabilmente un limite alla capacità del buffer.

In caso di buffer pieno: se si vuole mantenere immutata la semantica, occorre che il supporto a tempo di esecuzione provveda a **sospendere** il processo che invia il messaggio.

Comunicazione sincrona (rendez-vous semplice)

Semantica:
Il primo dei due processi comunicanti che esegue l'invio (mittente) o la ricezione (destinatario) **si sospende** in attesa che l'altro sia pronto ad eseguire l'operazione corrispondente.

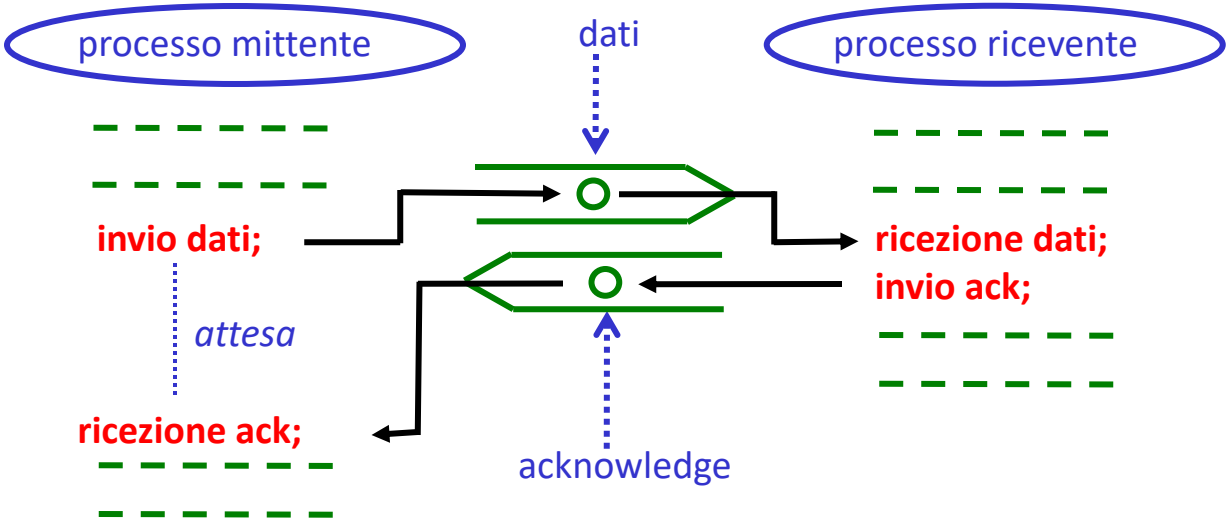
Effetti:
L'invio di un messaggio è un **punto di sincronizzazione**: ogni messaggio ricevuto contiene informazioni **attribuibili allo stato attuale** del processo mittente. Ciò semplifica la scrittura e la verifica dei programmi.

Non è necessaria l'introduzione di un **buffer**: un messaggio può essere inviato solo se il ricevente è pronto a riceverlo.

Comunicazione asincrona vs. sincrona

- L'invio con semantica **sincrona è più espressivo** (il contenuto dei messaggi ricevuti è **attuale**)
- La semantica **asincrona** consente di raggiungere **maggiore concorrenza/parallelismo.**
- La realizzazione del canale per nella comunicazione asincrona richiede opportuna struttura per l'accodamento dei messaggi.

Realizzazione di una comunicazione sincrona mediante comunicazioni asincrone



Comunicazione con sincronizzazione estesa ("rendez-vous" esteso)

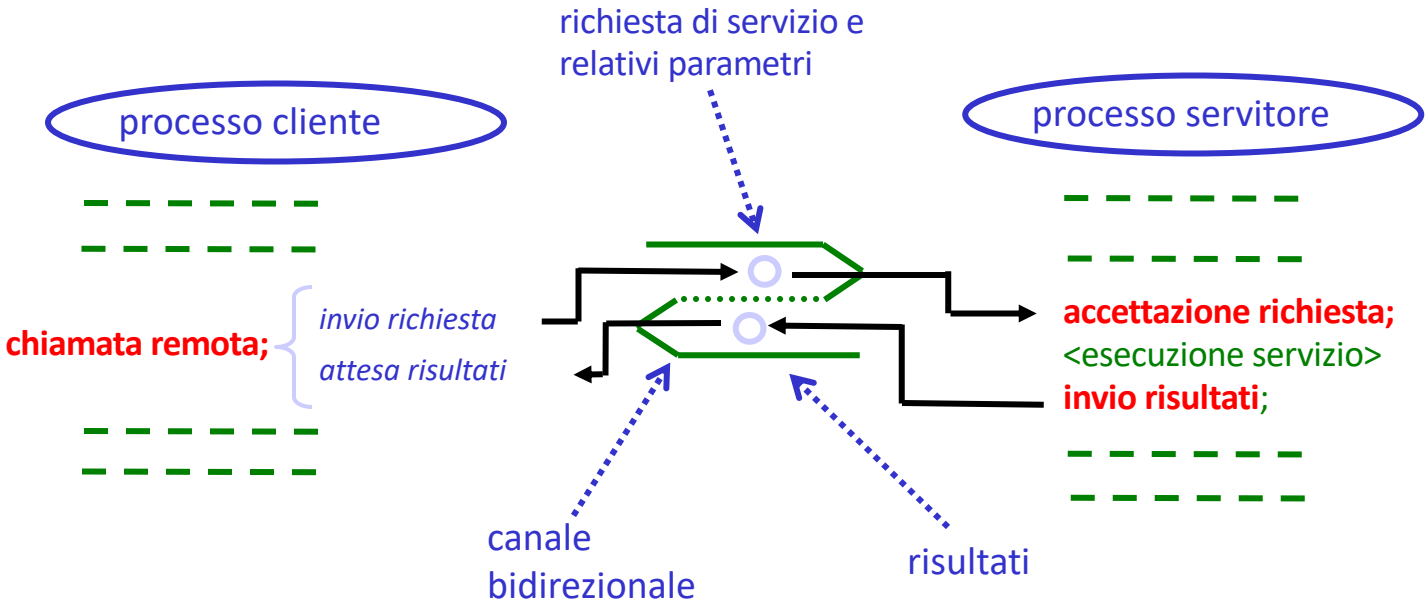
Assunzione: ogni messaggio inviato rappresenta una richiesta al destinatario dell'esecuzione di una certa azione.

Semantica:
Il processo mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta.

- Modello **cliente-servitore**.
- **Analogia** semantica con la **chiamata di procedura**.
- Riduzione di parallelismo.

Punto di sincronizzazione: semplificazione della verifica dei programmi.

Sincronizzazione estesa: chiamata di operazione remota



Costrutti linguistici e primitive per esprimere la comunicazione

Definizione del canale
Esempio: `port`

Un canale **port** identifica un canale asimmetrico **molti-a-uno**.

Dichiarazione di canale: port <tipo> <identificatore>;

```
port  int  ch1;
```

L'identificatore **`ch1`** denota un canale utilizzato per trasferire messaggi di tipo intero.

`port` viene dichiarato locale a un processo (il ricevente) ed è visibile ai processi mittenti mediante la dot notation:

<nome del processo>.<identificatore del canale>

Primitive di comunicazione: send

send è la primitiva che esprime l'**invio** di un messaggio:

send(<valore>) to <porta>;

- <**porta**> identifica in modo univoco il canale a cui inviare il messaggio (es: <nome processo>.<nome locale della porta>;)

- <**valore**> identifica una espressione dello stesso tipo di <porta> e rappresenta il contenuto del messaggio inviato:

Semantica: può essere

- **asincrona** -> canale bufferizzato
- **sincrona** -> canale a capacità nulla.

Primitive di comunicazione: send

Esempio:

```
port  int  ch1;
send(125) to P.ch1;
```

Il processo che la esegue invia il valore 125 al processo P tramite il canale ch1 da cui solo P può ricevere.

A seconda della semantica e delle caratteristiche del canale la send può sospendere o meno il processo che la esegue.

Ad esempio:

- **Send sincrona**: il processo attende che il destinatario esegua la primitiva di ricezione (receive) corrispondente;
- **Send asincrona**: il processo attende solo se il canale **`ch1`** è pieno.

Primitive di comunicazione: receive

Primitiva di ricezione:

P=**receive**(<**variabile**>) **from** <**porta**>;

<**porta**> identifica il **canale**, locale al processo ricevente, dal quale ricevere il messaggio ;

<**variabile**> è l'identificatore della variabile, dello stesso tipo di <porta>, a cui assegnare il valore del messaggio ricevuto.

Semantica:

- **Default**: Semantica **Bloccante**. La primitiva sospende il processo se non ci sono messaggi sul canale; quando c'è almeno un messaggio nel canale, ne estrae il primo e ne assegna a <variabile>il valore. La receive restituisce un valore del tipo predefinito process che identifica il nome del processo mittente.
- Alcuni linguaggi offrono **anche** una **semantica non bloccante**: se il canale è vuoto, il processo continua; se contiene almeno un messaggio, estrae il primo e lo assegna a <variabile>.

Primitiva receive

Esempio:

```
process proc;
proc = receive(m) from ch1;
```

Semantica bloccante:

Il processo che la esegue:

- si **sospende** se sul proprio canale ch1 non ci sono messaggi

altrimenti :

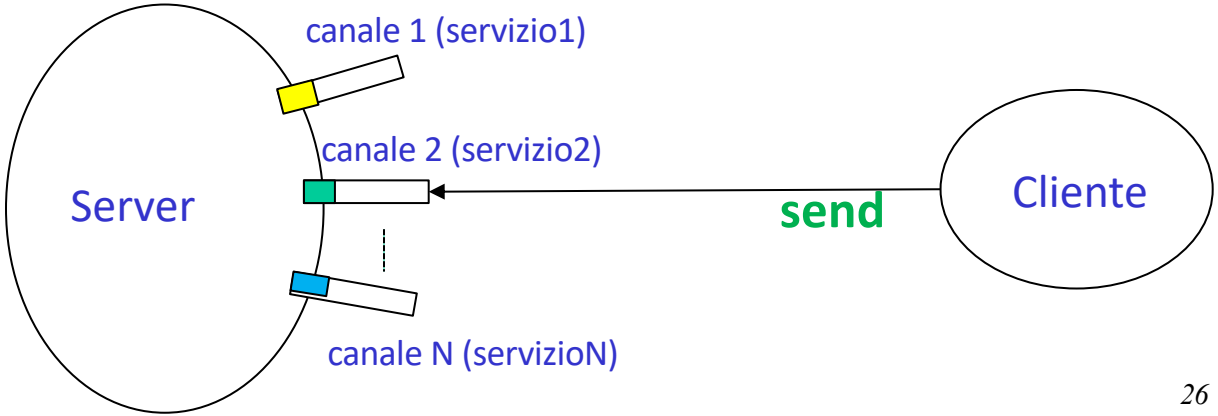
- **estrae il primo** messaggio da ch1 e ne assegna il valore a m;
- **assegna alla variabile proc l'identificatore del processo** che ha inviato il messaggio.

Receive bloccante e Modello Client-Server

Server: processo dedicato a servire le richieste di altri processi (clienti), ognuna rappresentata da un diverso messaggio.

In generale:

- il server ha la possibilità di eseguire diversi servizi, ognuno attivato in seguito alla ricezione di un messaggio di tipo diverso. A questo scopo il server gestisce **più canali di ingresso** (ad es. porte), ognuno dedicato alla ricezione delle richieste di un particolare servizio.



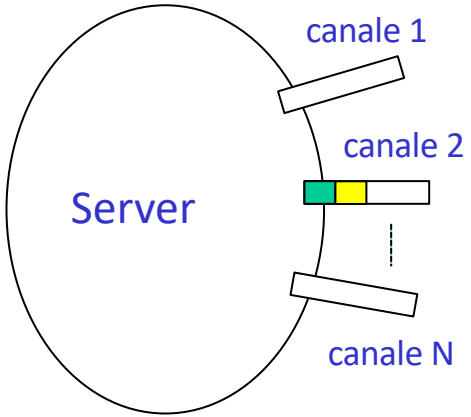
Receive bloccante e Client-Server

Problema:

Il server specifica il canale sul quale eseguire ogni receive. Se i canali sono più di uno, il server deve eseguire in sequenza le receive su ogni canale: poichè la receive ha semantica bloccante, c'è la possibilità di blocco su un canale mentre contemporaneamente ci sono messaggi in attesa di essere ricevuti su altri canali.

Esempio: una receive su canale1 (vuoto) **blocca** il server, nonostante vi siano richieste in attesa di essere ricevute su canale2.

```
// codice server:
...
while (true)
{           p = receive(m) from canale1;
            <esecuzione di servizio 1>
            p = receive(m) from canale2;
            <esecuzione di servizio 2>
            ...
}
```



Per risolvere il problema:

Receive con semantica non bloccante: verifica lo stato del canale, restituisce un messaggio (se presente) o un'indicazione di canale vuoto (non bloccante). Non sospende mai il processo che la esegue. Il server può eseguire un ciclo di ispezione/ricezione di/da tutti i canali.

In caso di presenza contemporanea di messaggi su più canali la scelta può essere fatta secondo diversi criteri (es. priorità, stato della risorsa), oppure in modo non deterministico.

Problema dell'**attesa attiva**: se tutti i canali sono vuoti, il server continua a iterare!

Meccanismo di ricezione ideale:

- consente al processo server di **verificare contemporaneamente la disponibilità di messaggi su più canali**;
- abilita la **ricezione di un messaggio da un qualunque canale contenente messaggi**;
- **quando tutti i canali sono vuoti, blocca il processo in attesa che arrivi un messaggio**, qualunque sia il canale su cui arriva,.

Questo meccanismo è realizzabile tramite i **comandi con guardia.**

Comando con guardia: sintassi

<guardia> -> <istruzione>;

dove **<guardia>** è costituita dalla **coppia**:

(<espressione booleana>; < receive>)

- L'<**espressione booleana**> viene detta **guardia logica**.
- La <**receive**> ha semantica **bloccante** e viene detta **guardia d'ingresso**.

Valutazione della guardia

<guardia> := (<espressione booleana>; < receive>)

La valutazione di una guardia può fornire tre diversi valori:

- **guardia fallita**: se **l'espessione booleana ha il valore false**.
- **guardia ritardata**: se **l'espressione booleana ha valore true** e nel canale su cui viene eseguita **non ci sono messaggi**.
- **guardia valida**: se **l'espressione booleana ha valore true** e nel canale **c'è almeno un messaggio** (-> la receive può essere eseguita senza ritardi)

Comando con guardia: semantica

<guardia> -> <istruzione>;

L'esecuzione di un comando con guardia determina i seguenti effetti:

La guardia viene valutata; si possono verificare 3 casi:

1. **guardia fallita**: il comando **fallisce** (-> non produce alcun effetto).
2. **guardia ritardata**: il processo che esegue il comando viene **sospeso**; quando arriverà il primo messaggio nel canare riferito dalla guardia d'ingresso, il processo verrà riattivato, **eseguirà la receive,** e successivamente l'<**istruzione**>.
3. **guardia valida**: il processo esegue la **receive** e successivamente l'<**istruzione**>.

Comando con guardia alternativo

Sintassi:

```
select
{       [ ] <guardia_1> -> <istruzione_1>;
        [ ] <guardia_2> -> <istruzione_2>;
        …
        [ ] <guardia_n> -> <istruzione_n>;
}
```

Rami del comando alternativo

Il comando con guardia alternativo (select) racchiude un numero arbitrario di comandi con guardia semplici .

Comando con guardia alternativo

Semantica:

Vengono valutate le guardie di tutti i rami. Si possono verificare **3 casi**:

1. se **una o più guardie sono valide** viene scelto, in maniera **non deterministica**, uno dei rami con guardia valida e la relativa guardia viene eseguita (cioè, viene eseguita la **receive**); viene quindi eseguita l'**istruzione** relativa al ramo scelto, e con ciò termina l'esecuzione dell'intero comando alternativo.
2. se **tutte le guardie non fallite sono ritardate**, il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da ritardata a valida e a quel punto procede come nel caso precedente.
3. se **tutte le guardie sono fallite** il comando termina.

Comando con guardia ripetitivo

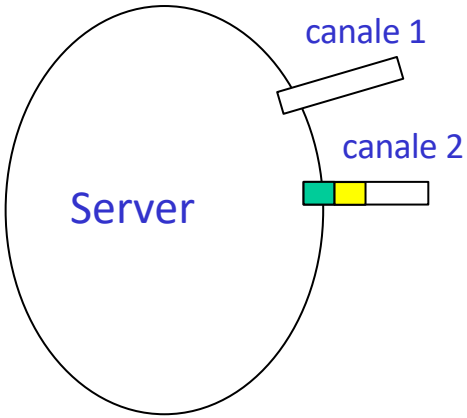
```
do
{       [ ] <guardia_1> -> <istruzione_1>;
        ....
        [ ] <guardia_n> -> <istruzione_n>;
}
```

Vengono valutate le guardie di tutti i rami. 3 casi:

1. **se una o più guardie sono valide** viene scelto, **in maniera non deterministica**, uno dei rami con guardia valida e la relativa guardia viene eseguita (viene cioè eseguita la receive); viene quindi eseguita l'istruzione relativa al ramo scelto, e poi si passa al ciclo successivo (ripetizione);
2. **se tutte le guardie non fallite sono ritardate**, il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da ritardata a valida; a quel punto procede come nel caso precedente (si passa all'iterazione successiva);
3. **se tutte le guardie sono fallite**, l'esecuzione del comando **termina**.

Comando con guardia e Modello Client-Server

Server: possibilità di ricezione di diverse richieste di servizio; più canali di ingresso ciascuno dedicato ad un tipo di richiesta.



Il comando con guardia ripetitivo (o alternativo) consente al server di ricevere il primo messaggio che arriva nell' insieme dei canali di ingresso.

Esempio di processo servitore

```
process server {
      port int  canale1;      //esecuzione di servizio1() su R
      port real canale2;      //esecuzione di servizio2() su R
     Tipo_di_R R; // risorsa gestita dal server
      int  x;
     real y;
     do
     {        [] (cond1); receive (x) from canale1; ->
              {       R.S1(x);
                       < eventuale restituzione dei risultati al cliente>;
              }
              [] (cond2); receive (y) from canale2; ->
              {       R.S2(y);
                       < eventuale restituzione dei risultati al cliente>;
              }
     }//fine do
}
```

Primitive di comunicazione asincrone

Primitive di comunicazione asincrone

Nel modello a scambio di messaggi, lo strumento di comunicazione di più basso livello è la **send asincrona**.

Accesso a risorse “condivise” -> processi servitori

Ad ogni operazione associata alla risorsa corrisponde un diverso servizio.

Soluzione di problemi di sincronizzazione con send asincrona

Prendiamo in considerazione alcuni tipici problemi di interazione nel modello a scambio di messaggi.

Data una risorsa, come implementare nel modello a scambio di messaggi (con **send asincrona**) il gestore della risorsa nel caso in cui sia previsto:

1. Una sola operazione
2. Più operazioni mutuamente esclusive
3. Più operazioni con condizioni di sincronizzazione

?

Obiettivo: individuare schemi di soluzioni ai diversi problemi basati sulla send asincrona e confrontarli con le soluzioni dello stesso problema nel modello a memoria comune usando il **monitor**.

1. Esempio: Risorsa condivisa con una sola operazione

Risorsa condivisa che mette a disposizione di un insieme di processi "clienti" una sola operazione con il solo vincolo della mutua esclusione.

Soluzione nel modello a memoria comune:

il gestore è un **monitor** con **una operazione entry**:

```
monitor gestore
{    tipo_var var;
     <eventuale inizializzazione>
     entry tipo_out fun (tipo_in x)
     {    <corpo della funzione fun>;
     }
}

gestore ris; //istanza del gestore
```

```
thread client{
tipo_in a;
tipo_out b;
….
b=ris.fun(a);
...
}
```

Soluzione es.1 nel modello a scambio di messaggi:

La risorsa viene gestita da un **processo server** che offre un unico servizio senza condizioni di sincronizzazione.

Cliente: un solo canale risposta di tipo tipo-out

```
process cliente {
 port tipo_out risposta;
 tipo_in  a;
 tipo_out  b;
 process p;
    ……
    ……
 send(a)to server.input;
 p=receive(b)from risposta;
    ……
}
```

Server: un solo canale input di tipo tipo_in

```
tipo_out fun(tipo_in x);
process  server {
  port tipo_in input;
  tipo_var  var;
  process p;
  tipo_in  x;
  tipo_out y;
      <eventuale inizializzazione>;
      while(true){
          p=receive(x)from input;
          y =fun(x);
          send(y)to p.risposta;
       }
}
```

2. Esempio: Risorsa "condivisa" con più operazioni

Risorsa condivisa che mette a disposizione di un insieme di processi clienti due operazioni con il solo vincolo della mutua esclusione.

Soluzione nel modello a memoria comune:

Il gestore è un **monitor** con **due operazioni entry**:

```
monitor gestore{
tipo_var var;
...
entry tipo_out1 fun1 (tipo_in1 x1) {
<corpo della funzione fun1>; }

entry tipo_out2 fun2 (tipo_in2 x2) {
<corpo della funzione fun2>; }
}
                    ...
gestore ris;
```

```
thread client{
tipo_in1 a1;
tipo_out1 b1;
tipo_in2 a2;
tipo_out2 b2;
...
b1=ris.fun1(a1);
...
b2=ris.fun2(a2);
...
}
```

Soluzione nel modello a scambio di messaggi:

La risorsa è gestita da un processo servitore che offre 2 servizi senza condizioni di sincronizzazione.

Soluzione senza comandi con guardia: Un solo canale per entrambi i tipi di richiesta.

```
typedef struct{
        enum (fun1,fun2)servizio;
        union{
                tipo_in1 x1;// parametri fun1
                tipo_in2 x2;// parametri fun2
        }parametri;
        }in_mess; /* tipo del messaggio */
```

```
tipo_out1 fun1 (tipo_in1 x1); tipo_out2 fun2 (tipo_in2 x2);

process server {
        port in_mes input;
        tipo_var var; process p; in_mes richiesta; tipo_out1 y1;
        tipo_out2 y2;
        while (true) {
                p=receive (richiesta) from input;
                switch (richiesta.servizio) {
                case fun1: {y1=fun1(richiesta.parametri.x1);
                                send(y1) to p.risposta1;
                                break; }
                case fun2: {y2=fun2(richiesta.parametri.x2);
                            send(y2) to p.risposta2;
                              break; }
                        }
                }
}
```

```
process cliente {
 port tipo_out1 risposta1;
 port tipo_out1 risposta1;
 tipo_in1  a1;
 tipo_in2  a2;
 in_mes M;
 tipo_out1  b1;
 tipo_out2  b2;
 process p;
 <inizializzazione M, in base al servizio scelto>
 if (<servizio1>) then
        {       send(M)to server.input;
                p=receive(b1)from risposta1;
        }
 else   {       send(M)to server.input;
                p=receive(b2)from risposta2;
        }
  ..
}
```

Questa soluzione ha una **scarsa scalabilità**: la generalizzazione a N servizi introduce complessità nella dichiarazione del tipo in_mess (N alternative nella union).

Soluzione con comandi con guardia: **per ogni servizio** il server apre un **canale distinto**.

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process  server {
   port tipo_in1   input1; //canale per le richieste relative a fun1
   port tipo_in2   input2; //canale per le richieste relative a fun2
   tipo_var  var;
   process  p;
   tipo_in1  x1;  tipo_in2  x2;
   tipo_out1  y1;  tipo_out2  y2;
   {< eventuale inizializzazione>;}
   do
   {     [] p = receive (x1) from input1; ->
                  y1=fun1(x1);
                  send (y1) to p.risposta1;
         [] p = receive (x2) from input2; ->
                  y2=fun2(x2);
                  send (y2) to p.risposta2;
    }
}
```

```
process cliente {
 port tipo_out1 risposta1;
 port tipo_out1 risposta1;
 tipo_in1  a1;
 tipo_in2  a2;
 tipo_out1  b1;
 tipo_out2  b2;
 process p;

<inizializzazione a1 o a2>

 if (<servizio1>) then
        {       send(a1)to server.input1;
                p=receive(b1)from risposta1;
        }
 else   {       send(a2)to server.input2;
                p=receive(b2)from risposta2;
        }
  ……
}
```

3. Risorsa condivisa con più operazioni e condizioni di sincronizzazione

La risorsa è gestita da un processo servitore che offre due operazioni con condizioni di sincronizzazione.

Soluzione nel modello a memoria comune:

La risorsa è gestita da un monitor con 2 entry e 2 variabili condizione:

```
condition c1,c2;
```

```
entry tipo_out1 op1 (tipo_in1 x1)
{
        ……
        if(!cond1)  wait (c1);
        ……
        signal (c2);
        …….
}
```

```
entry tipo_out2 op2 (tipo_in2 x2)
{
        ……
        if(!cond2)  wait (c2);
        ……
        signal(c1);
        …….
}
```

(HP: politica «signal and wait»).

Soluzione nel modello a scambio di messaggi:

La risorsa è gestita da **un processo servitore con tanti servizi quante sono le operazioni previste:**

- viene associato ad ogni servizio un canale distinto;
- la ricezione delle richieste in arrivo viene realizzata con un **comando con guardia** ripetitivo, con due rami (uno per ogni servizio); in ogni ramo la **guardia logica** esprime la **condizione di sincronizzazione**.

Es: 2 operazioni:

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process  server {
   port tipo_in1   input1;
   port tipo_in2   input2;
   tipo_var  var;
   tipo_in1  x1;  tipo_in2  x2;
   tipo_out1  y1;  tipo_ou2t  y2;
   ...
   do{
    [] (cond1); p = receive (x1) from input1; ->
            { y1=fun1(x1); send (y1) to p.risposta1;}
     [] (cond2); p = receive (x2) from input2; ->
            { y2=fun1(x2); send (y2) to p.risposta2;}
   }
}
```

Esempi

Esempi di processi servitori: gestore di un pool di risorse equivalenti

Ogni risorsa è gestita da un processo.

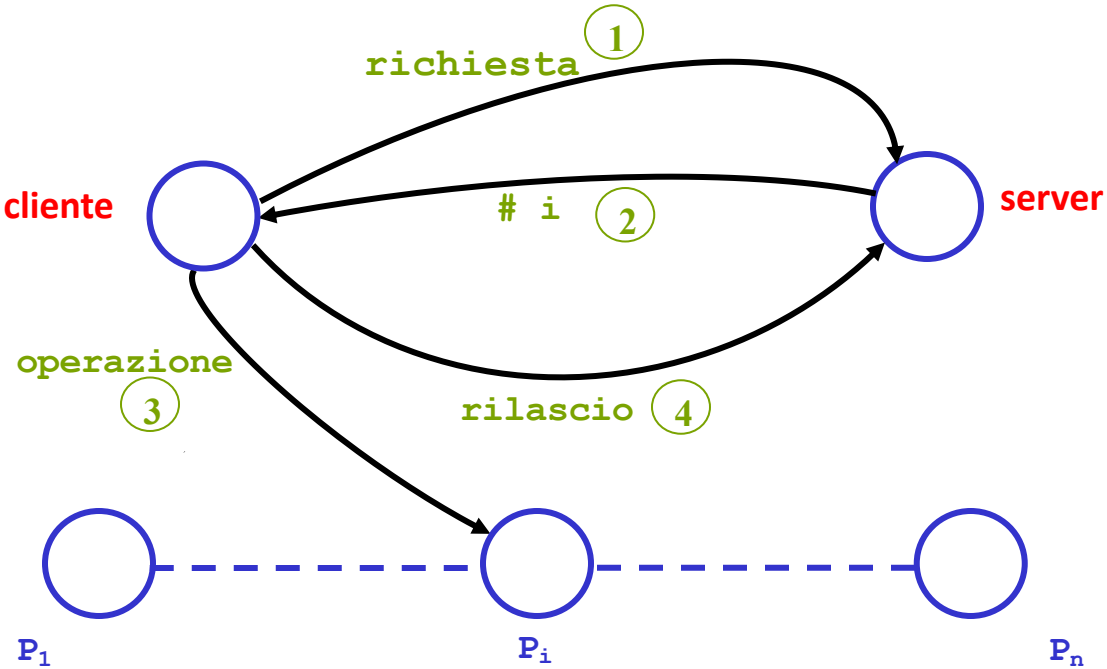
n risorse R1, R2, ...Rn => **n processi gestori**: P1,P2,..Pn.

Processo **server (gestore del pool)**: nell'allocazione di una risorsa Ri, indica al processo cliente richiedente il processo P_i che gestisce la risorsa.

Il **cliente** invia al server una richiesta per ottenere l'indice di una risorsa disponibile. Il server restituisce l'indice i del processo che gestisce la risorsa disponibile; successivamente il cliente interagisce direttamente con P_i per richiedere l'esecuzione di operazioni sulla risorsa Ri.

Quando il client avrà terminato di utilizzare la risorsa Ri, invierà al server un messaggio per indicare il «rilascio» di Ri.

Gestore di un pool di risorse equivalenti: protocollo



processi gestori delle risorse del pool

Pool di risorse equivalenti: cliente

```
process cliente{
      port int risorsa;
      signal s;
      int r;
      process p, server;
      …….
      send(s) to server.richiesta;// richiesta di una risorsa
      p=receive (r) from risorsa;//attesa della risposta
      <uso delle risorsa r-sima>
      send (r) to server.rilascio;//rilascio della risorsa
      ...
 }
```

Pool di risorse equivalenti: server

```
process server{
    port signal richiesta; // la richiesta è un messaggio privo di
                           //contenuto informativo (tipo signal)
    port int rilascio;
    int disponibili=N;
    boolean  libera[N];
    process p ;
    signal s;
    int  r;
    for (int i=0; i<N; i++)libera[i]=true; //inizializzazione
    do
    {   [] (disponibili>0); p= receive(s)from richiesta; ->
                int i=0;
                while (!libera[i]) i++;
                libera[i] = false;
                disponibili--;
              send (i) to p.risorsa;
       [] p=receive(r)from rilascio; ->
                disponibili++;
                libera[r] = true;
     }
}
```

Specifica di strategie di priorità

Si consideri un server che gestisce un **pool di risorse** adottando una politica basata su **priorità**.

Criterio di priorità: il server deve privilegiare le richieste di P0 rispetto a quelle di P1 e queste rispetto a quelle di P2….

Pertanto:

- Quando un qualunque processo Pi invia al server una richiesta , questa viene servita se ci sono risorse disponibili, mentre , in caso contrario, Pi rimane sospeso.

- Quando una risorsa viene rilasciata, il server deve essere in grado di riconoscere se ci sono processi clienti sospesi e, in questo caso, quali sono.

- Tra i clienti sospesi, va scelto quello a priorità più alta (indice più basso).

Pool di risorse con priorità: cliente

```
process cliente{
   port int risorsa;
    signal s;
    int r;
       …….
     send(s) to server.richiesta;
     p=receive (r) from risorsa;
     <uso delle risorsa r-sima>
     send (r) to server.rilascio;
     ...
}
```

Pool di risorse con priorità: server

```
process server{
    port signal richiesta;
    port int rilascio;
    int disponibili=N;
    boolean  libera[N];
    process p ;
    signal s;
    int  r;
    int sospesi=0; boolean bloccato[M];
    process client[M];
    { //inizializzazione
     for (int i=0; i<N; i++) libera[i]=true;
     for (int j=0; j<M; j++) bloccato[j]=false;
    client[0]="P0";..... client[M-1]="PM-1";
    }
```

```
process cliente{
    port int risorsa;
   signal s;
   int r;
      ……
    send(s) to server.richiesta;
    p=receive (r) from risorsa;
    <uso delle risorsa r-sima>
    send (r) to server.rilascio;
    ...
}
```

```
    do
    {   [] p=receive(s)from richiesta; ->
              if (disponibili>0 {
                 int i=0;
                 while (!libera[i]) i++;
                 libera[i] = false;
                 disponibili--;
                 send (i) to p.risorsa;}
              else { int j=0; sospesi++;
                          while(client[j]!=p) j++;
                          bloccato[j]=true;}
      [] p:= receive (r) from rilascio; ->
              if (sospesi == 0) {
                          disponibili++;
                          libera[r] = true; }
              else {      int i=0;
                          while (!bloccato[i]) i++;
                          sospesi --;
                          bloccato[i] = false;
                          send (r)to client[i].risorsa;}
    }
}
```

Esempio: produttori & consumatore

(scambio di dati molti a uno)

Si vuole realizzare la soluzione al problema produttori-consumatori, nel caso in cui vi siano:

- più produttori
- uno ed un solo consumatore

L'introduzione di un processo server è superflua in quanto la soluzione è offerta direttamente dalla porta a cui un produttore invia i messaggi ed il produttore li preleva: la porta implementa il buffer «condiviso».

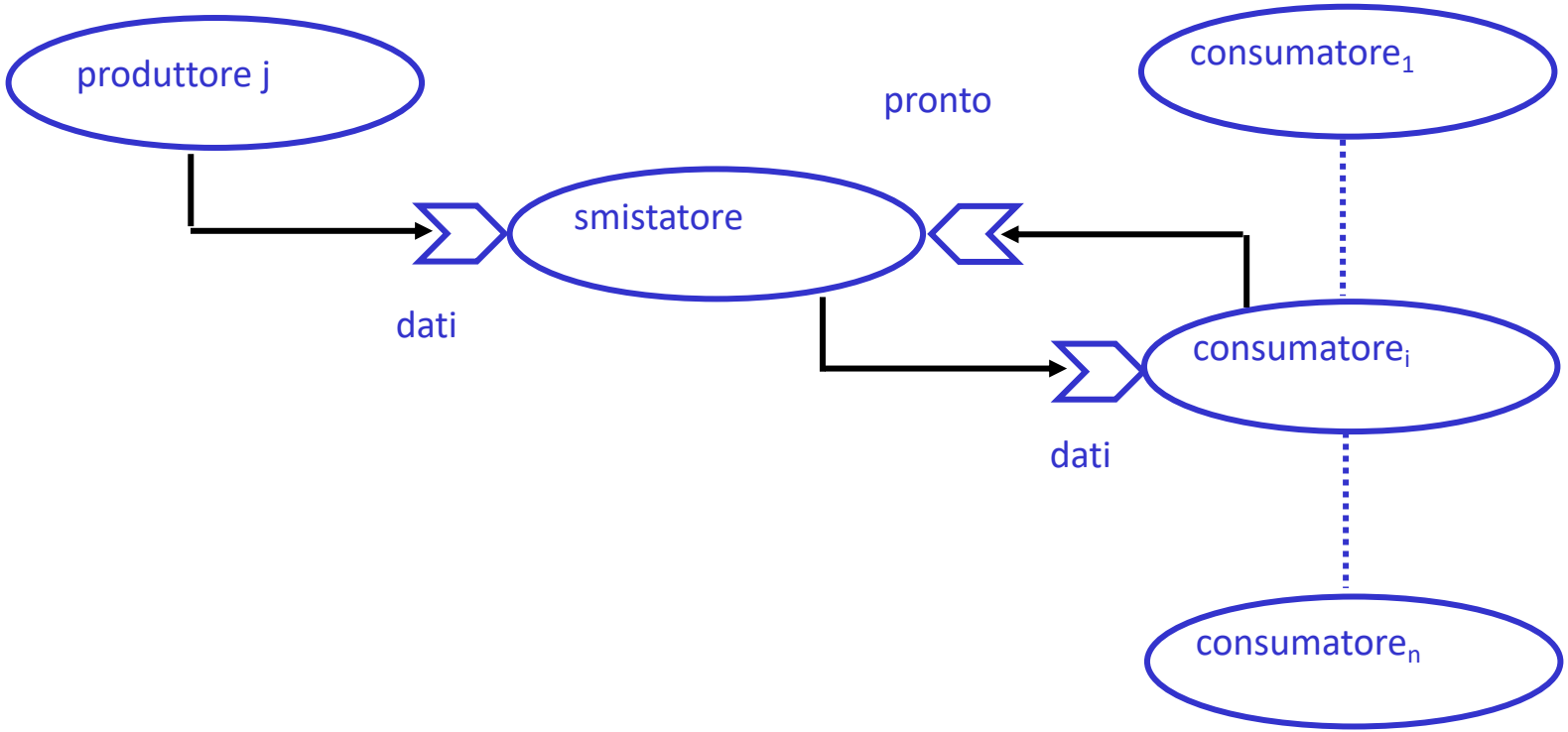
Nel caso in cui si prevedano più processi consumatori (molti-a-molti):

E' necessario interporre tra produttori e consumatori un **processo server** (***smistatore***), a cui i produttori inviano i messaggi.

Per ogni messaggio M ricevuto, lo smistatore ha con il compito di scegliere il consumatore a cui trasmettere M, in base alla disponibilità a ricevere messaggi che i singoli consumatori comunicano al server.

L'uso della **send asincrona** elimina la necessità di introdurre esplicitamente un buffer in quanto i canali di comunicazione contengono già, al loro interno, un buffer di dimensione teoricamente illimitata.

Esempi di processi servitori: produttori e consumatori (scambio di dati molti-a-molti)



Il processo «**smistatore**» ha due porte:

- **dati** per ricevere dati dai produttori
- **pronto** per ricevere il messaggio inviato dai consumatori pronti a ricevere il messaggio. Il messaggio inviato dal consumatore è privo di contenuto informativo (messaggio di tipo signal).

Se T è il tipo generico dei messaggi inviati dai produttori:

- La porta dati del server è di tipo T.
- La porta dati del consumatore è di tipo T.

Quando desidera ricevere un messaggio, ogni consumatore invia un messaggio sulla porta pronto del server e si mette in attesa di ricevere il messaggio sulla propria porta dati.

```
process smistatore{
    port T dati;
    port signal pronto;
    T messaggio;
    process prod, cons;
    signal s;
    while (true) {
        cons=receive(s)from pronto;
        prod=receive(messaggio)from dati;
        send(messaggio)to cons.dati;
    }
}
```

NB: la dimensione del buffer è implicitamente determinata dalla capacità del canale **dati**.

Esempi di servitori: mailbox (molti-a-molti) con capacità limitata

Limite superiore alla dimensione del buffer dei messaggi: i messaggi inviati, ma non ancora ricevuti non possono essere più di **N**.

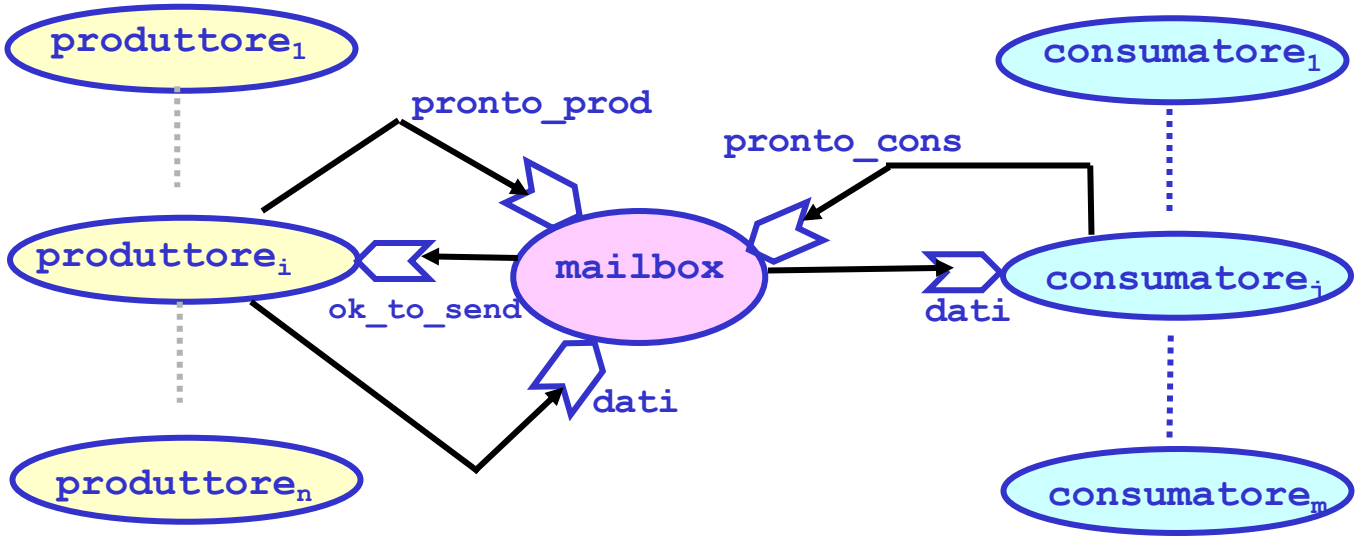
Vincoli:

- Il produttore non può inviare un nuovo messaggio se il buffer è pieno (cioè, i precedenti N messaggi non sono stati ancora ricevuti); in tal caso attende.

  -> prevediamo la porta "**pronto_prod**" per la verifica della condizione di inserimento (tramite una guardia)

- Il consumatore non può estrarre un messaggio se il buffer è vuoto; in tal caso attende.

  -> prevediamo la porta "**pronto_cons**" per la verifica della condizione di estrazione(tramite una guardia)


produttore1
produttorei
produttoren
pronto_prod
ok_to_send
mailbox
dati
pronto_cons
dati
consumatore1
consumatorej
consumatorem

```
process mailbox{
    port T dati; // implementa il buffer
    port signal pronto_prod, pronto_cons;
    T messaggio;
    process prod, cons;
    signal s;
    int contatore=0;
    do
    {   [](contatore<N);prod=receive(s)from pronto_prod;->
              {       contatore ++;
                      send(s)to prod.ok_to_send; }
        [] (contatore>0);cons=receive(s)from pronto_cons;->
               {      prod=receive(messaggio)from dati;
                      contatore--;
                      send(messaggio)to cons.dati; }
    }
}
```

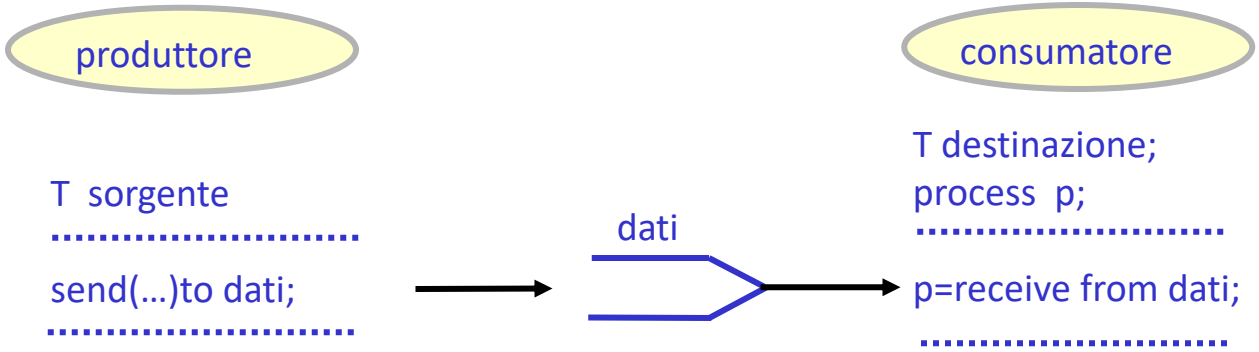
```
process produttore_i {
        port signal ok_to_send;
        T messaggio; process p; signal s;
        ...
        <produci il messaggio>;
        send(s) to mailbox.pronto_prod;
        p=receive (s) from ok_to_send;
        send (messaggio) to mailbox.dati; }

process consumatore_j {
        port T dati;
        T messaggio; process p;
        signal s;
        ...
        send(s) to mailbox.pronto_cons;
        p=receive (messaggio) from dati;
        <consuma il messaggio>; }
```

Primitive di comunicazione sincrone

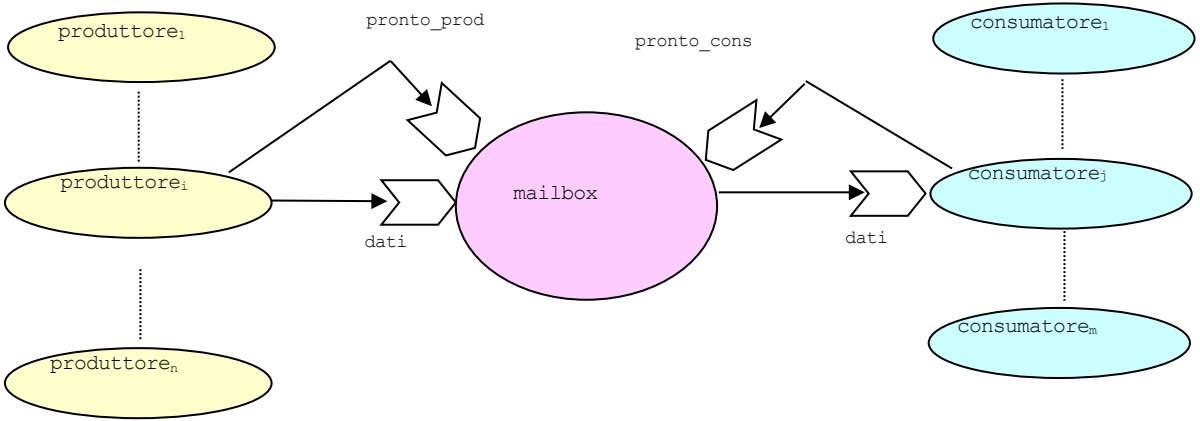
Confronto fra le primitive asincrone e le primitive sincrone

- Minore grado di concorrenza delle primitive sincrone rispetto alle asincrone
- Con le primitive sincrone non è più necessaria la presenza di buffer nei canali di comunicazione

Scambio di dati:
singolo produttore-singolo consumatore



Mailbox di dimensioni finite (protocolli simmetrici)



- Rispetto al caso della send asincrona: il canale non è in grado di bufferizzare i messaggi inviati -> necessità di una struttura interna al processo mailbox: coda di messaggi

- Nel descrivere la soluzione, per semplicità, non viene dettagliata la realizzazione del tipo astratto coda_messaggi di cui, locale al processo mailbox, viene dichiarata l'istanza coda. In particolare, supporremo che su oggetti del tipo coda_messaggi si possa operare con le seguenti funzioni:

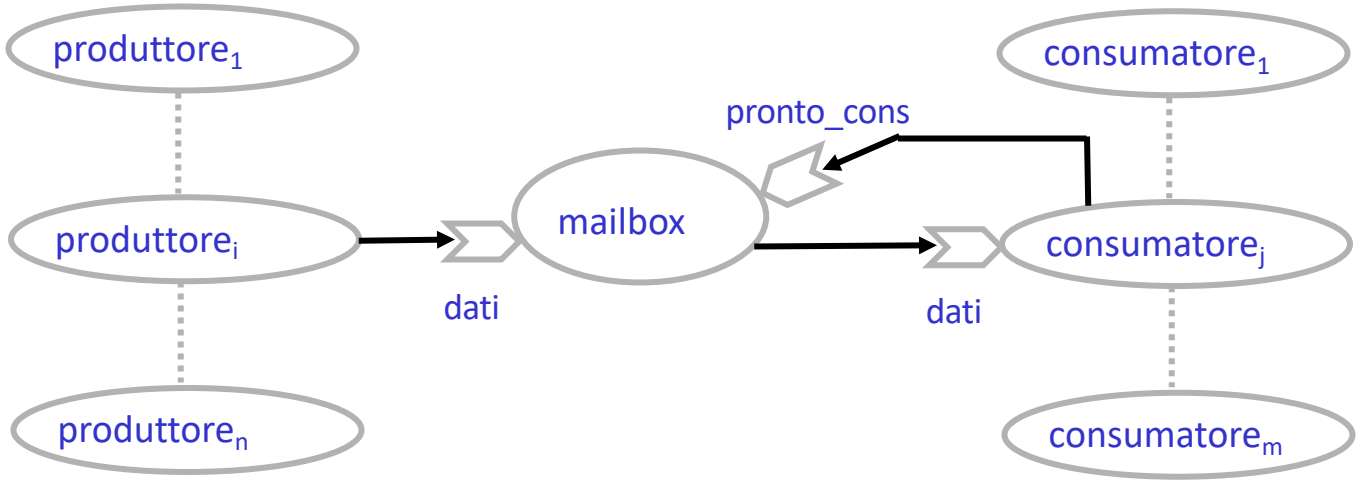
```
void inserimento(T mes);  //per inserire mes nella coda
T estrazione(); //restituisce il primo elemento della coda
boolean piena();  // restituisce true se la coda è piena
boolena vuota();  // restituisce true se la coda è vuota
```

```
process mailbox{
        port T dati;
        port signal pronto_prod, pronto_cons;
        T messaggio;
        process p;
        signal s;
        coda_messaggi  coda;
        <inizializzazione>;
        do {
        [] (!coda.piena()); p=receive(s)from pronto_prod;
                ->      p=receive(messaggio)from dati;
                        coda.inserimento(messaggio);
        [](!coda.vuota()); p=receive(s)from pronto_cons;
                ->      messaggio = coda.estrazione;
                        send(messaggio)to p.dati;
        }
}
```

```
process produttore_i{
        T messaggio;
        signal s;
                ...
        <produci il messaggio>;
        send(s)to mailbox.pronto_prod;
        send(messaggio)to mailbox.dati;
                ...
}

process consumatore_j{
        port T dati;
        T messaggio;
        process p;
        signal s;
                ...
        send(s)to mailbox.pronto_cons;
        p=receive(messaggio)from dati;
        <consuma il messaggio>;
                ...
}
```

Mailbox di dimensioni finite

è possibile ottimizzare il protocollo (lato produttore):



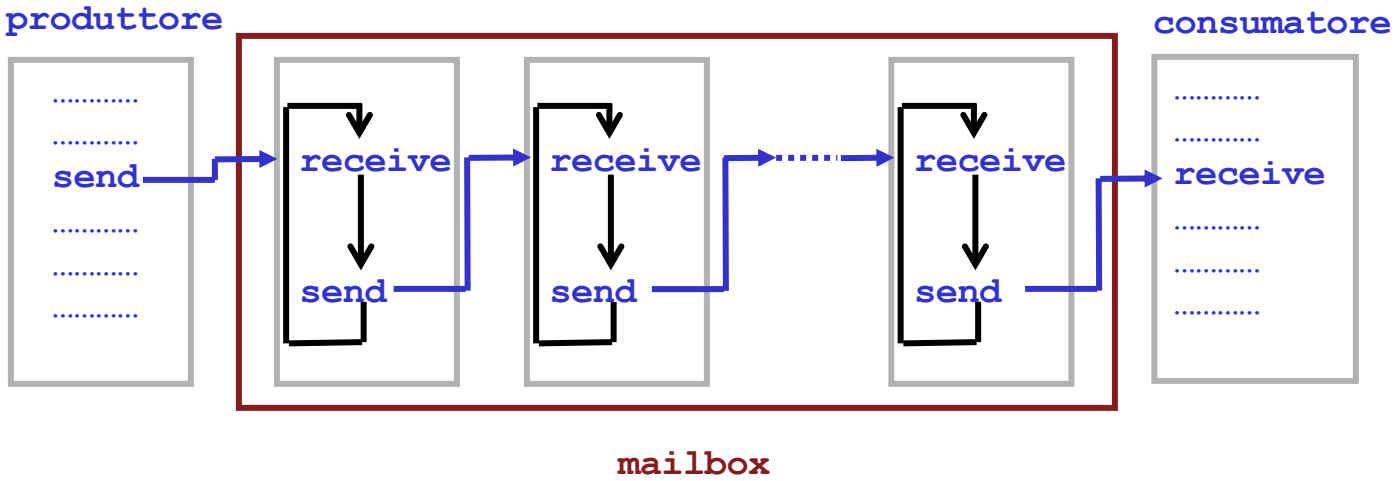
```
process mailbox{
        port T dati;
        port signal pronto_cons;
        T messaggio;
        process p;
        signal s;
        coda_messaggi coda;
        <inizializzazione>;
        do {
        [] (!coda.piena()); p=receive(messaggio)from dati;
                ->      coda.inserimento(messaggio);
        [] (! coda.vuota()); p=receive(s)from pronto_cons;
                ->      messaggio = coda.estrazione;
                        send(messaggio)to p.dati;
        }
}
```

```
process produttore_i{
        T messaggio;
        process p;
        signal s;
        ……
        <produci il messaggio>;
        send(messaggio)to mailbox.dati;
        ……
}
```

Mailbox concorrente

Un buffer di dimensione N può essere implementato da un insieme di processi concorrenti collegati secondo uno schema a cascata (pipeline).

Ogni processo ha una struttura ciclica e, sfuttando la semantica sincrona della send, può trattenere un singolo messaggio inserito, ma non ancora estratto.



Struttura processo i-simo nella pipeline:

```
process m_i{ //(0≤i≤N-2)
        port T dati;
        T buffer;
        process p;
        while (true){
                p=receive(buffer) from dati;
                send(buffer) to m_i+1.dati;}
        }
```

Specifica di strategie di priorità (es: pool con 1 sola risorsa)

```
process server{
    port signal richiesta;
    port int rilascio;
    boolean  libera;
    process p ;
    signal s;
    int sospesi=0; boolean bloccato[M];
    process client[M]

    { //inizializzazione
    libera=true;
    for (int j=o; j<M; j++) bloccato[j]=false;
    client[0]="P0";.....client[M-1]="PM-1";
    }
```

```
process cliente{
    port int risorsa;
   signal s;

      .....
    send(s) to server.richiesta;
    p=receive (s) from risorsa;
    <uso delle risorsa r-sima>
    send (s) to server.rilascio;
    ...
}
```

```
do {
[] p = receive(s)from richiesta; ->
          if (libera) {
             libera = false;
             send (s) to p.risorsa;}
       else {
             sospesi++;
             int j=0;
             while(client[j]!=p) j++;
             bloccato[j]=true;}
[] p = receive (s) from rilascio; ->
      if (sospesi == 0) libera= true;
      else {
             int i = 0;
             while (!bloccato[i]) i++;
             sospesi --;
             bloccato[i] = false;
             send (s)to client[i].risorsa;}
}
```

Realizzazione dei meccanismi di comunicazione

Realizzazione delle primitive asincrone

Realizzazione delle primitive di comunicazione e del costrutto port (per definire i canali utilizzati dalle primitive) utilizzando gli strumenti di comunicazione offerti dal nucleo del sistema operativo.

Riferimento alle primitive asincrone in quanto più "primitive": infatti, le primitive di tipo sincrono possono essere tradotte dal compilatore del linguaggio in termini di primitive asincrone.

Realizzazione in Architetture mono/multielaboratore e architetture distribuite.

Architetture mono e multielaboratore

Ipotesi (semplificative):

1. Tutti i messaggi scambiati tra i processi sono di un **unico tipo T** predefinito a livello di nucleo (es., stringa di byte di dimensione fissa).

2. Tutti i canali sono **da molti ad uno** (**port**) e quindi associati al processo ricevente

3. Essendo le primitive asincrone, ogni porta deve contenere un buffer (coda di messaggi) **di lunghezza indefinita**

```
typedef struct {
    T  informazione;
    PID  mittente;
    messaggio *  successivo;
} messaggio;
```

```
typedef struct {
    messaggio * primo;
    messaggio * ultimo;
} coda_di_messaggi;
```

```
void inserisci(messaggio * m, coda_di_messaggi c)
{       if (c.primo == null) c.primo = m;
        else c.ultimo -> successivo = m;
        c.ultimo = m;
        m -> successivo = null;
}
```

```
messaggio * estrai(coda_di_messaggi  c)
{       messaggio *  pun;
        pun = c.primo;
        c.primo = c.primo -> successivo;
        if (c.primo == null) c.ultimo = null;
        return pun;
}
```

```
boolean  coda_vuota (coda_di_messaggi  c) {
         if (c.primo == null) return true;
         return false;
}
typedef struct {
       coda_di_messaggi  coda;
       p_porta  puntatore;
 } des_porta;

typedef des_porta *p_porta;
typedef struct {
         p_porta porte_processo[M];
         PID nome;
         modalità_di_servizio  servizio;
         tipo_contesto  contesto;
         tipo_stato  stato;
         PID  padre;
         int  N_figli;
         des_figlio prole[max_figli];
         p_des successivo;
} des_processo;
```

Descrittore del processo

Il campo `stato` registra se il processo è attivo oppure bloccato; in questo caso tiene traccia delle porte sulle quali è in attesa.

```
boolean  bloccato_su(p_des p, int ip) {
     <testa il campo stato nel descrittore del processo di cui p è il puntatore e
     restituisce il valore true se il processo risulta bloccato in attesa di ricevere messaggi
     dalla porta il cui indice nel campo porte_processo è ip >;
}
```

```
void blocca_su(int ip) {
     <modifica il campo stato del descrittore del
     processo_in_esecuzione
     per indicare che lo stesso si blocca in attesa di messaggi dalla porta il cui indice nel
     campo porte_processo è ip >;
}
```

```
void testa_porta (int ip){/*verifica la presenza di messaggi*/
     p_des esec = processo_in_esecuzione;
     p_porta pr = esec->porte_processo[ip];
     il (coda_vuota(pr->coda)) { /* sospensione*/
          blocca_su(ip);
          assegnazione_CPU; // context switch
     }
}
```

```
messaggio *estrai_da_porta (int ip) {
     messaggio *m;
     p_des esec = processo_in_esecuzione;
     p_porta pr = esec->porte_processo[ip];
     m = estrai(pr->coda);
     return m;
}
```

```
void inserisci_porta (messaggio * m, PID proc, int ip){
     p_des destinatario = descrittore(proc);
     p_porta pr = destinatario->porte_processo[ip];
     inserisci(m, pr->coda);
     if (bloccato_su(destinatario, ip)) attiva(destinatario);
}
```

[**attiva(p)** porta il processo p nello stato di ready]

```
void send (T inf, PID proc, int ip){
     messaggio *m=new messaggio;
     m -> informazione = inf;
     m -> mittente = processo_in_esecuzione;
     inserisci_porta(m, proc, ip);
}
```

```
void receive (T *inf, PID *proc, int ip){
     messaggio *m;
     testa_porta(ip);
     m=estrai_da_porta(ip);
     *proc=m->mittente;
     *inf=m->informazione;
}
```

Ricezione su più canali (es. comandi con guardia):

necessita` di verificare la presenza di messaggi su piu` porte

```
int  testa_porte(int ip[], int n){
     p_porta pr; int ris=-1; int indice_porta;
     p_des esec=processo_in_esecuzione;
     for (int i=0; i<n; i++){
          indice_porta=ip[i];
          pr=esec->porte_processo[indice_porta];
          if(coda_vuota(pr->coda)) blocca_su(indice_porta);
          else{
               ris=indice_porta;
               esec->stato= <processo attivo>;
               break;
          }
     }
     if(ris==-1) assegnazione_CPU(); //tutte le porte vuote: sospensione
     return ris;
}
```

In questo caso la scansione procede in ordine di indice, quindi verrà selezionata la porta non vuota con l'indice minimo. Per realizzare una selezione non deterministica, l'ordine di scansione deve essere random.

```
int receive_any(T *inf, PID *proc, int ip[], int n){
    messaggio * mes; int indice_porta;
    do
        indice_porta=testa_porte(ip,n);
    while(indice_porta==-1);
    mes = estrai_da_porta(indice_porta);
    proc = &(mes->mittente);
    inf =&(mes ->informazione);
    return indice_porta;
}
```

Architetture distribuite

Sistemi operativi distribuiti (DOS- Distributed Operating System):

Insieme di nodi tra loro **omogenei** e tutti dotati dello **stesso sistema operativo** (stesso nucleo).

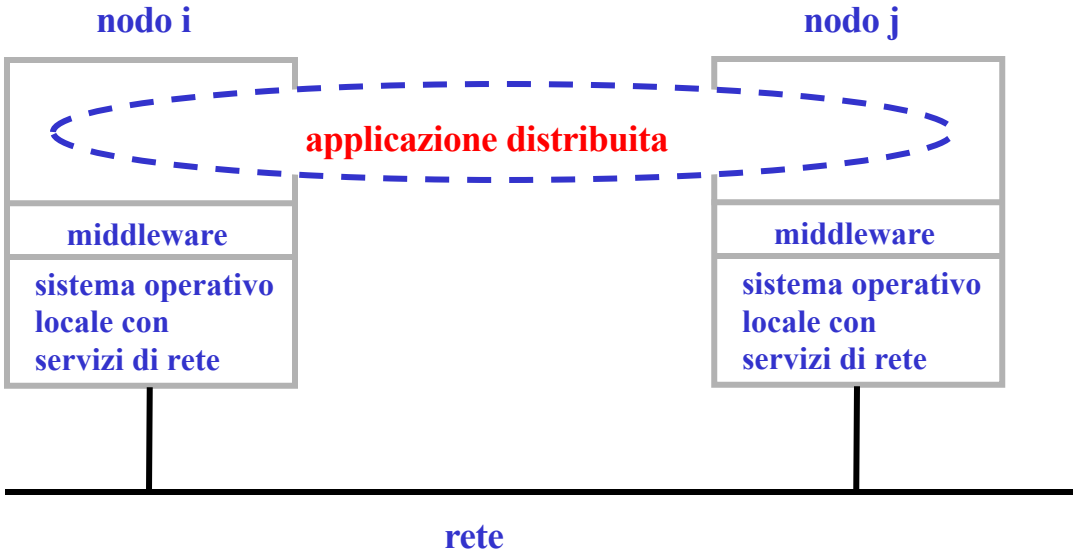
Scopo del sistema: gestire tutte le risorse nascondendo all'utente la loro distribuzione sulla rete.

Sistemi Operativi di Rete (NOS- Network operating Systems):

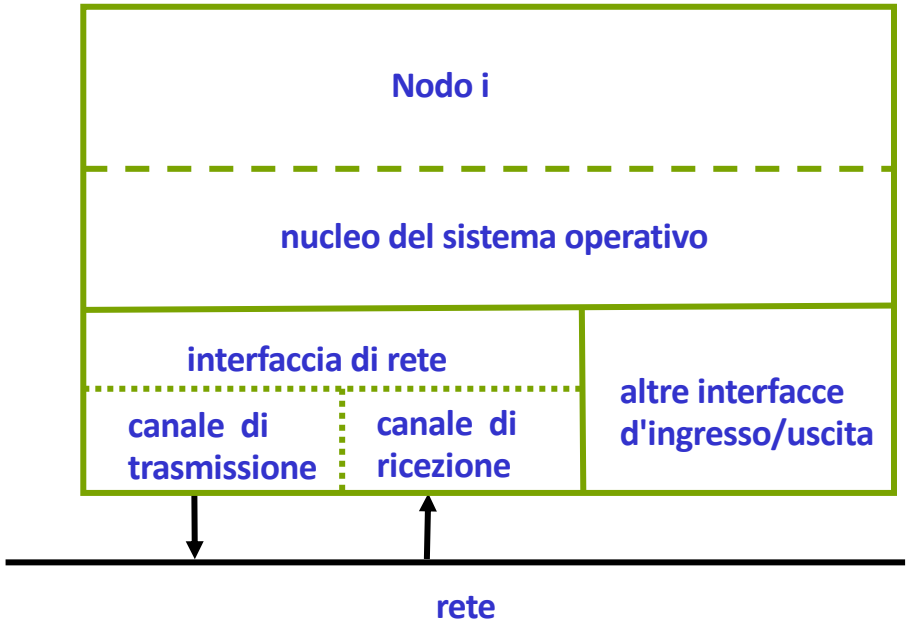
Insieme di nodi eterogenei, con sistemi operativi diversi e autonomi, nodo per nodo.

Ogni nodo della rete è in grado di offrire servizi a clienti remoti presenti su altri nodi della rete (es. uso di socket).
Trasparenza della distribuzione delle risorse viene ottenuta mediante il **middleware** (interposto su ogni nodo tra S.O. e le applicazioni).

Architetture distribuite
Network Operating System



Sistemi DOS
Interfaccia di rete



Pacchetti, interfacce, canali

L'unità di trasmissione tra nodi è il **pacchetto**. La struttura del pacchetto dipende dalla realizzazione (cioè dai protocolli di comunicazione adottati dalla rete).

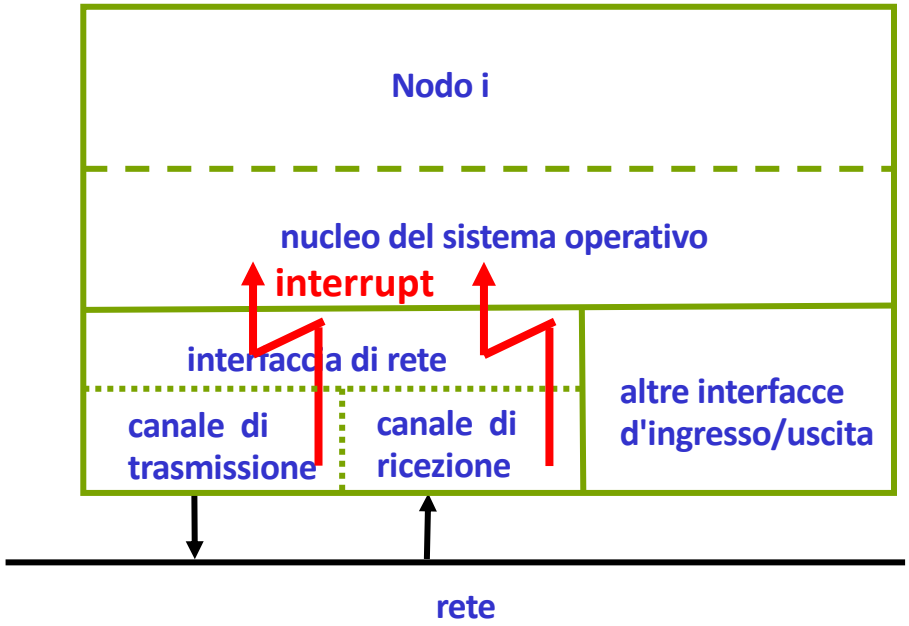
L'interfaccia di rete è strutturata in **2 parti** (**canali**): uno per la **trasmissione** dei messaggi, uno per la **ricezione**.

Il **canale di trasmissione** viene acceduto per realizzare l'invio di pacchetti; ad esso sono associati:

- una **coda di pacchetti**, nella quale ogni processo sender deposita il proprio pacchetto se il canale è occupato da un altro processo.
- un registro **buffer** della dimensione di un pacchetto.

Il **canale di ricezione** viene acceduto per realizzare la ricezione di pacchetti; ad esso è associato un registro **buffer**, nel quale viene depositato ogni pacchetto inviato al nodo.

Arrivo/partenza di pacchetti verso/da canali di ricezione/trasmissione vengono notificati tramite **interruzioni**.


Nodo i
nucleo del sistema operativo
interrupt
interfaccia di rete
canale di trasmissione
canale di ricezione
altre interfacce d'ingresso/uscita
rete

Il pacchetto è l'informazione che viene trasmessa attraverso il canale: contiene, oltre al messaggio (inf, mittente), anche le informazioni relative al processo destinatario e alla porta di destinazione. La struttura del pacchetto dipende dalla realizzazione.

```
void invia_pacchetto (packet p) {
        if (<canale di trasmissione occupato>)
                packet_queue.inserisci(p);
        else {
                <inserimento di p nel registro buffer del canale>;
                <attivazione trasmissione>;//al termine:interruzione
        }
}
```

```
void tx_interrupt_handler() {
        packet p;
        salvataggio_stato();
        if (!packet_queue.vuota()){
                p = packet_queue.estrai();
                <inserimento di p nel registro buffer del canale>;
                 <attivazione trasmissione>;
        }
        ripristino_stato();
}
```

```
typedef struct {
          int indice-nodo;
          int PID_locale;
} PID;
```

```
void send(T inf, PID proc, int ip){
          if (proc.indice_nodo == nome_nodo)
                    local_send(inf, proc, ip);
          else remote_send(inf, proc, ip);
}
```

```
void remote_send(T inf, PID proc, int ip){
          packet p;
          PID mit=processo_in_esecuzione;
          int indice_nodo_destinatario= proc.indice_nodo;
          <vengono riempiti i vari campi del pacchetto p: in particolare, viene
           inserito nel campo relativo al nodo a cui inviare il pacchetto il valore
           indice_nodo_destinatario. Inoltre, nel campo del pacchetto destinato
           a contenere le informazioni da inviare vengono inseriti il nome mit del processo
           che invia, e i tre parametri della funzione inf, proc, e ip>;
           invia_pacchetto(p);
}
```

```
void rx_interrupt_handler() {
   packet p;
   PID mit; T inf; PID proc; int ip;
   salvataggio_stato();
   <assegnamento a p del pacchetto ricevuto presente nel buffer del canale>;
   <attivazione ricezione>;
   <estrazione dal campo del pacchetto p, contenente le informazioni
    ricevute, del nome mit del mittente e dei tre parametri della funzione
    send inf, proc, e ip>;
   messaggio * m=new messaggio;
   m -> informazione = inf;
   m -> mittente = mit;
   inserisci_porta(m, proc, ip);
   ripristino_stato();
}
```

Realizzazione delle primitive sincrone mediante primitive asincrone

In un nucleo nel quale siano implementate le send asincrone, è possibile costruire le primitive sincrone nel modo seguente:

```
void send (T inf, PID proc, int ip){
        signal s;
        a_send(inf, proc, ip);
        a_receive(s, proc, ak);
}
```

```
void receive (T &inf, PID &proc, int ip){
        signal s;
        a_receive (inf, proc, ip);
        a_send (s, proc, ak);
}
```

NB: a_send, a_receive sono send e receive con semantica asincrona

Realizzazione “nativa” delle primitive sincrone: implementazione di send sincrona e receive come primitive di nucleo

```
typedef struct {
     T  informazione;
     PID  mittente;
} messaggio;
```

```
typedef  struct {
     messaggio buffer[N];/* N num. massimo mittenti*/
     int primo, ultimo, cont;
} coda_di_N_messaggi;
```

```
typedef struct {
     coda_di_N_messaggi  coda;
     p_porta successivo;
} des_porta;
```

```
typedef des_porta  *p_porta;
```

```
void inserisci(messaggio  m, coda_di_N_messaggi c){
   //inserisce il messaggio m nella coda di messaggi c:
  in particolare un messaggio proveniente da Pi viene assegnato a buffer[i]
.....}
```

```
messaggio  estrai (coda_di_N_messaggi  c) {
   //estrae dalla coda di messaggi c  un messaggio e lo restituisce
.....  }
```

```
boolean  coda_vuota (coda_di_N_messaggi  c) {
   //testa la coda di messaggi c  per verificare se il suo buffer è vuoto
.....
            }
```

```
boolean  bloccato_su(p_des p, int ip) {
    <testa il campo stato nel descrittore del processo di
     cui p è il puntatore e restituisce il valore true se il
     processo risulta bloccato in attesa di ricevere messaggi
dalla porta il cui indice nel campo porte_processo
          è ip >;
}
```

```
void blocca_su(int ip) {
     <modifica il campo stato del descrittore del
      processo_in_esecuzione per indicare che lo
      stesso si blocca in attesa di messaggi dalla porta il cui
      indice nel campo porte_processo è ip >;
}
```

```
void testa_porta (int ip){
// testa la porta di indice ip del processo in esecuzione bloccandolo se vuota
     p_des esec = processo_in_esecuzione;
     p_porta pr = esec->porte_processo[ip];
     if (coda_vuota(pr->coda)) {
          blocca_su(ip);
          assegnazione_CPU;
     }
}
```

```
void inserisci_porta (messaggio mes, PID proc, int ip){
     /*inserisce il messaggio mes nella porta di indice ip del processo proc e, se
       questo è in attesa sulla porta , lo attiva*/
     p_des destinatario = descrittore(proc);
     p_porta pr = destinatario->porte_processo[ip];
     inserisci(m, pr->coda);
     if(bloccato_su(destinatario,ip)) attiva(destinatario);
}
```

```
messaggio estrai_da_porta (int ip) {
     /*estrae dalla porta di indice ip (porta sicuramente non vuota)
     del processo in esecuzione un messaggio, lo restituisce e attiva il
     mittente del messaggio ricevuto */
     messaggio  mes; p_des mit;
     p_des esec = processo_in_esecuzione;
     p_porta pr = esec->porte_processo[ip];
     mes = estrai(pr->coda);
     mit=descrittore(mes.mittente);
     attiva(mit);
     return mes;
}
```

```
void attendi_ricezione() {
     p_des esec = processo_in_esecuzione;
     esec->stato = <bloccato sulla send>;
     assegnazione_CPU();
}
```

```
void send (T inf, PID proc, int ip){
    messaggio  mes;
    mes.informazione = inf;
    mes.mittente = processo_in_esecuzione;
    inserisci_porta(mes, proc, ip);
    attendi_ricezione();
}
```

```
void receive (T &inf, PID &proc, int ip){
    messaggio  mes;
    testa_porta(ip);
    mes=estrai_da_porta(ip);
    proc=mes.mittente;
    inf=mes.informazione;
}
```