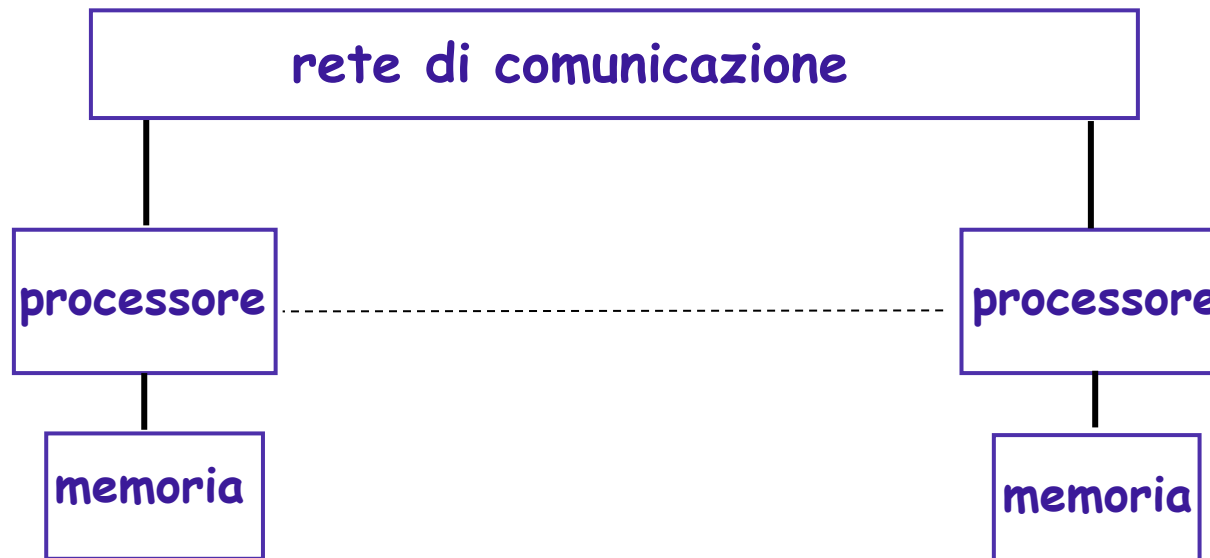


Modello a scambio di messaggi

- Aspetti caratterizzanti il modello
- Canali di comunicazione
- Primitive di comunicazione

Aspetti caratterizzanti il modello

modello architetturale di macchina (virtuale) concorrente



Aspetti caratterizzanti il modello

- Ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria (virtuale) locale
- Ogni risorsa del sistema è accessibile ad un solo processo
- Se una risorsa è necessaria a più processi applicativi, ciascuno di questi (processi clienti) dovrà richiedere all'unico processo che può operare sulla risorsa (processo server) di eseguire sulla stessa l'operazione richiesta e restituirgli i risultati dell'operazione

Aspetti caratterizzanti il modello

- In questo modello architetturale di macchina concorrente il concetto di gestore di una risorsa coincide con quello di processo server
- cambia quindi il paradigma di programmazione utilizzato per consentire ad un processo di usufruire dei servizi offerti da una risorsa
- il meccanismo di base utilizzato dai processi per qualunque tipo di interazione è costituito dal meccanismo di scambio di messaggi

Canali di comunicazione

- Il concetto fondamentale nel modello a scambio di messaggi è quello di canale, inteso come collegamento logico mediante il quale due processi comunicano.
- È compito del nucleo del sistema operativo fornire l'astrazione "canale" come meccanismo primitivo per lo scambio di informazioni (rete di comunicazione nel caso di un sistema distribuito o memoria comune nel caso di sistemi multiprocessori).
- È compito di un linguaggio di programmazione offrire gli strumenti linguistici di alto livello per consentire al programmatore di specificare i canali di comunicazione e di utilizzarli per programmare le varie interazioni tra i processi dell'applicazione.

Parametri che caratterizzano il concetto di canale

- a) la **tipologia del canale**, intesa come direzione del flusso dei dati che un canale può trasferire;
- c) la **designazione del canale** e dei processi **sorgente e destinatario** di ogni comunicazione;
- c) il tipo di **sincronizzazione** fra i processi comunicanti.

Tipi di canale con riferimento alla direzione del flusso dei dati

a) Canali monodirezionali

prevedono il flusso di dati in una sola direzione (mittente verso ricevente).

b) Canali bidirezionali

usati sia per inviare che per ricevere informazioni (es. cliente-servitore).

Tipi di canale con riferimento alla designazione dei processi comunicanti

a) **link** canale simmetrico:

da-uno-a-uno

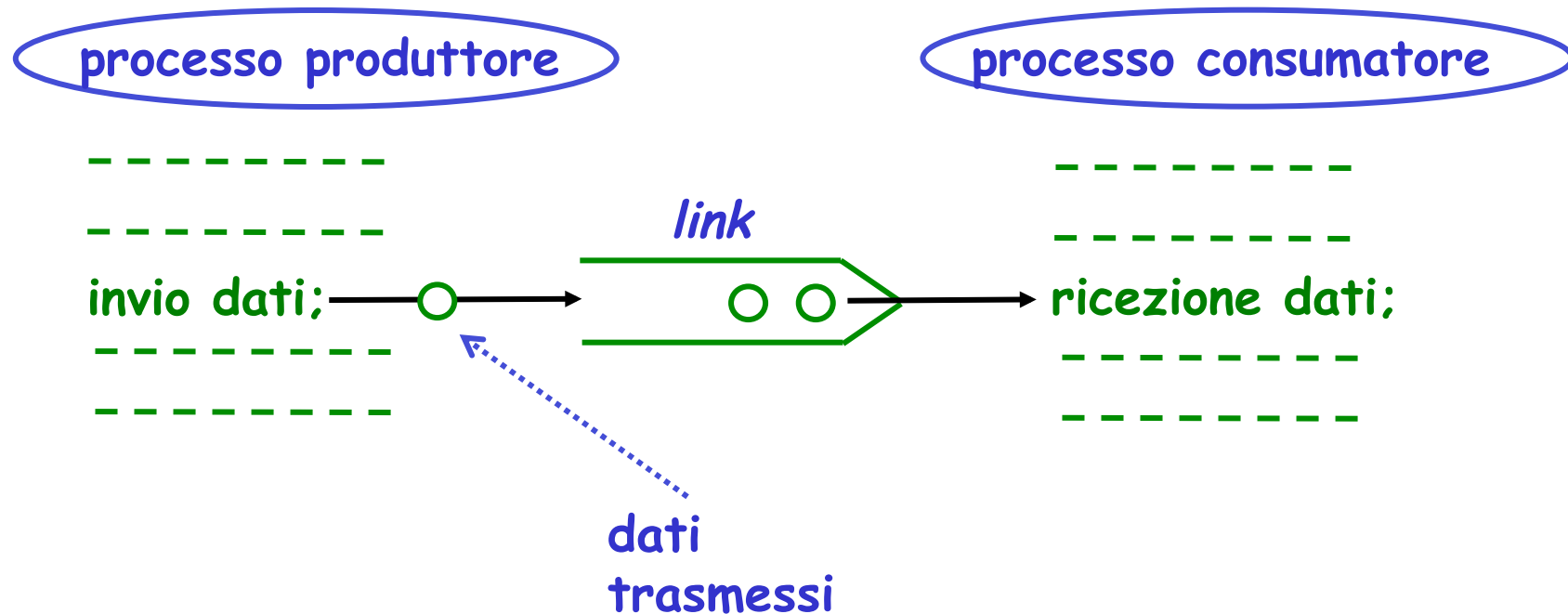
b) **port** canale asimmetrico:

da-molti-a-uno

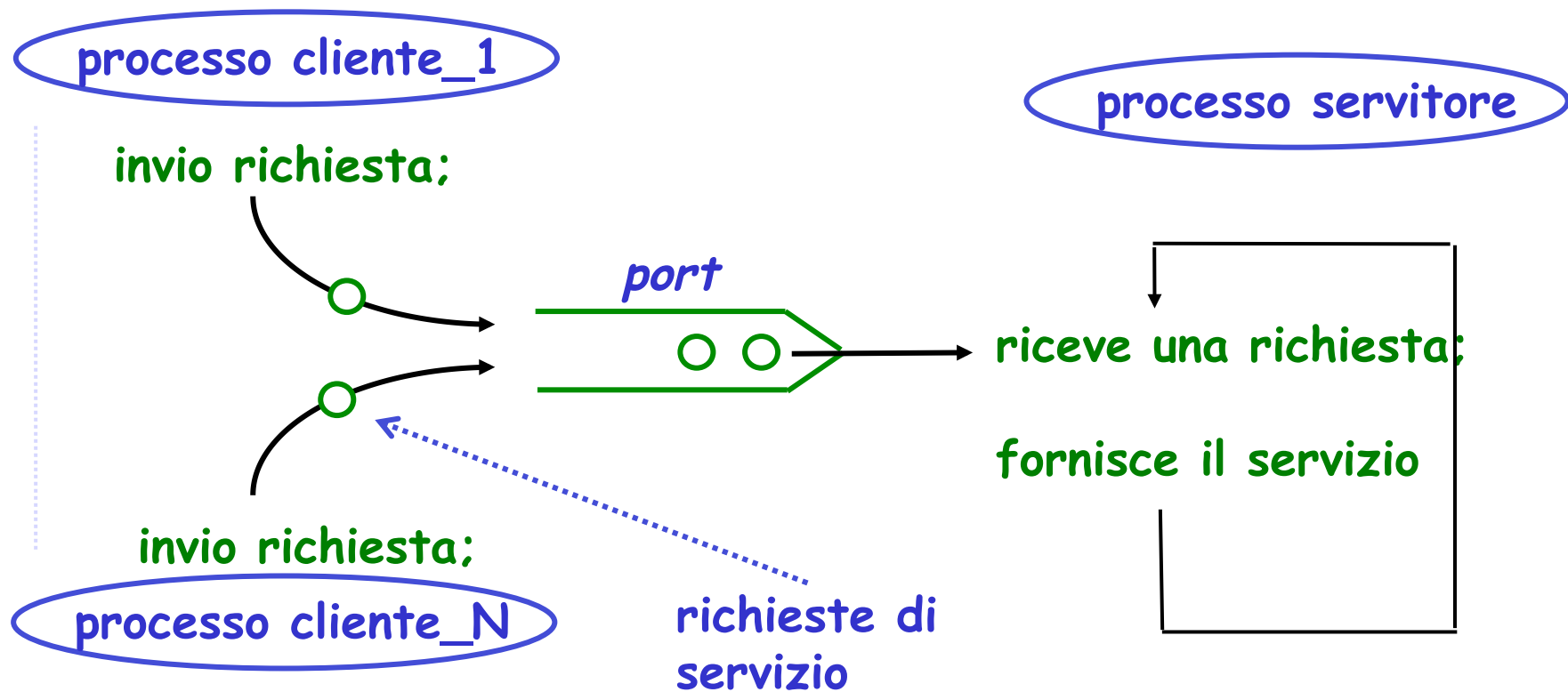
c) **mailbox** canale asimmetrico:

da-molti-a-molti

Comunicazione simmetrica produttore-consumatore



Comunicazione asimmetrica: es. cliente-servitore



Tipi di canale con riferimento alla tipologia di sincronizzazione

a) Comunicazione asincrona

c) Comunicazione sincrona

c) Comunicazione con sincronizzazione estesa

Comunicazione asincrona

- Il processo mittente **continua la sua esecuzione** immediatamente dopo che il messaggio è stato inviato.
- Il messaggio ricevuto contiene informazioni che non possono essere associate allo stato attuale del mittente (difficoltà nella verifica dei programmi).
- **La send non è un punto di sincronizzazione.** E' necessario pertanto prevedere, per realizzare un particolare schema di interazione, uno **scambio esplicito di messaggi** che non contengono informazioni da elaborare.

Comunicazione asincrona

- Carenza espressiva (difficoltà di verifica dei programmi) vs. grado di concorrenza.
 - Richiede un buffer di capacità illimitata.
 - Ogni implementazione prevede un limite alla capacità del buffer.
- In caso di **coda piena**:

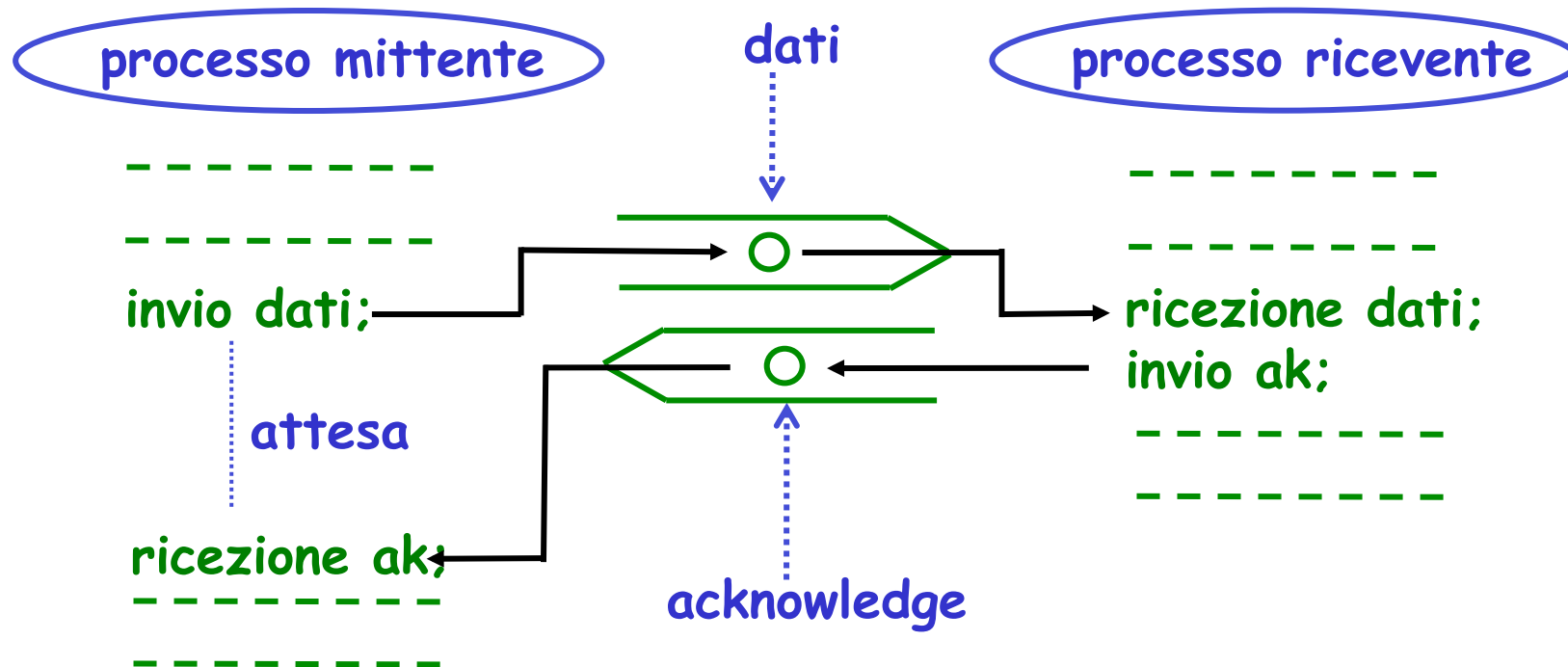
- Se si vuole mantenere immutata la semantica della send, occorre che il supporto a tempo di esecuzione provveda a sospendere il processo;

In alcune implementazioni, ciò non è previsto e l'operazione di invio può terminare in modo anomalo generando un'eccezione quando il buffer è pieno

Comunicazione sincrona (rendez-vous semplice)

- Il primo dei due processi comunicanti che esegue o l'invio o la ricezione **si sospende** in attesa che l'altro sia pronto ad eseguire l'operazione duale.
- Non esiste la necessità dell'introduzione di un buffer; un messaggio può essere inviato solo se il ricevente è pronto a riceverlo.
- Un messaggio ricevuto contiene informazioni **corrispondenti allo stato attuale** del processo mittente. Ciò semplifica la scrittura e la verifica dei programmi.

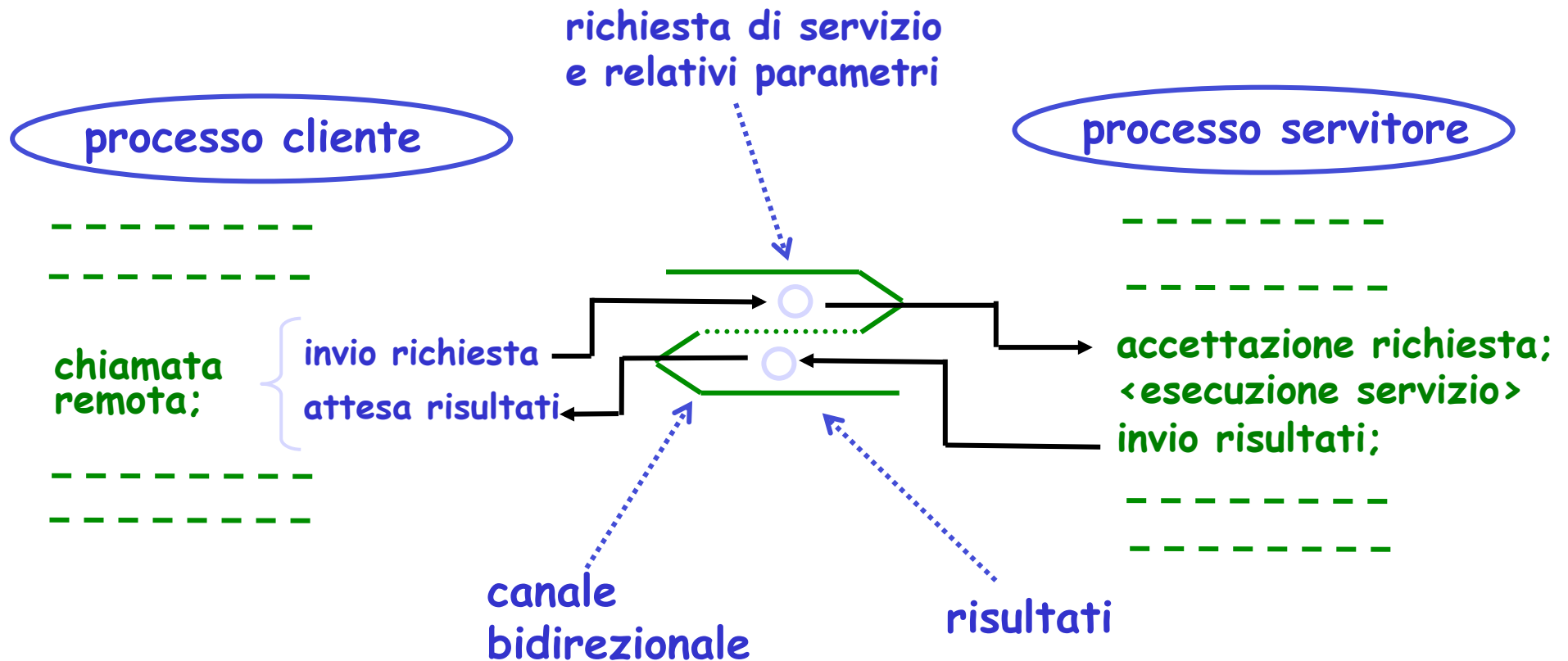
Simulazione di una comunicazione sincrona mediante comunicazioni asincrone



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

- Il processo mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta.
- Analogia semantica con la chiamata di procedura.
- Modello cliente-servitore.
- Riduzione di parallelismo.
- Punto di sincronizzazione: semplificazione della verifica dei programmi.

Schema relativo a una chiamata di operazione remota



Primitive di comunicazione

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

```
port int ch1;
```

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

- Un canale dichiarato mediante il costrutto `port` identifica un canale asimmetrico da-molti-a-uno. 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

Primitiva di invio:

`send(<valore>) to <porta>;`

- `<porta>` identifica in modo univoco il canale a cui inviare il messaggio (normalmente espresso come `<nome processo>.<nome locale della porta>;`
- `<valore>` identifica una espressione dello stesso tipo di `<porta>` e il cui valore rappresenta il contenuto del messaggio inviato:

```
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.

Primitive di comunicazione

Primitiva di ricezione:

`receive(<variabile>) from <porta>;`

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

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

- La primitiva **sospende il processo** se non ci sono messaggi sul canale e restituisce un valore del tipo predefinito **process** che identifica il nome del processo mittente.

Primitiva receive

Es:

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

Il processo che la esegue si sospende se sul proprio canale ch1 non ci sono messaggi ;

altrimenti :

- estrae il primo di loro e ne assegna il valore a m;
- assegna alla variabile proc, di tipo process, il nome del processo che ha inviato il messaggio.

- Possibilità di ricezione di diverse richieste di servizio:
più canali di ingresso ciascuno dedicato
ad un tipo di richiesta.

- Scelta del canale sul quale eseguire la receive.
Possibilità di blocco su un canale in presenza di
messaggi su un altro:

➔ primitiva **receive non bloccante** che verifica lo stato di un
canale, restituisce un messaggio se presente o
un'indicazione di canale vuoto (non bloccante)

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

Problema dell'attesa attiva.

Meccanismo di ricezione ideale:

- consente al processo server di verificare la disponibilità di messaggi sui due (o piu`) canali;
- abilita la ricezione di un messaggio da uno qualunque di essi contenente messaggi;
- blocca il processo in attesa che arrivi un messaggio, qualunque sia il canale su cui arriva, quando nessun canale contiene messaggi.

Comando con guardia: sintassi

<guardia> -> <istruzione>;

<guardia> è costituita da una coppia:

(<espressione booleana>; <primitiva receive>)

Una guardia viene valutata e può fornire tre diversi valori:

“guardia **fallita**”: se l'espressione booleana ha il valore false.

“guardia **ritardata**”: se l'espressione booleana ha valore true, ma la **receive** è bloccante poichè sul canale su cui viene eseguita non ci sono messaggi pronti.

“guardia **valida**”: se l'espressione booleana ha valore true e la **receive** può essere eseguita senza ritardi

Comando con guardia: semantica

<guardia> -> <istruzione>;

La guardia viene valutata; 3 casi:

1. guardia **fallita**: il comando fallisce (non produce alcun effetto).
2. guardia **ritardata**: il processo che esegue il comando viene sospeso; quando arriva il primo messaggio il processo viene riattivato, esegue la receive e successivamente esegue <istruzione>.
3. guardia **valida**: la receive viene eseguita ed il processo esegue <istruzione>.

Comando con guardia alternativo

```
if
    [] <guardia_1> -> <istruzione_1>;
...
[] <guardia_n> -> <istruzione_n>;
fi
```

- a) vengono valutate le guardie di tutti i rami;
- b1) 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 contenuta nella guardia scelta); viene quindi eseguita l'istruzione relativa al ramo scelto e con ciò termina l'esecuzione dell'intero comando alternativo.
- b2) 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;
- b3) se tutte le guardie sono fallite il comando termina.

Comando con guardia ripetitivo

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

- a) vengono valutate le guardie di tutti i rami;
- b1) 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 contenuta nella guardia scelta); viene quindi eseguita l'istruzione relativa al ramo scelto e, successivamente, l'esecuzione dell'intero comando viene ripetuta.
- b2) 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;
- b3) se tutte le guardie sono fallite l'esecuzione del comando termina.

Esempio di processo servitore

```
process server {  
  port int  servizioA; //esecuzione di A() su R  
  port real servizioB; //esecuzione di B() su R  
  Tipo_di_R R;  
  int  x;  
  real y;  
  do  
    [] (condA); receive (x) from servizioA; ->  
      { R.A(x);  
        <eventuale restituzione dei risultati al cliente>;  
      }  
    [] (condB); receive (y) from servizioB; ->  
      { R.B(y);  
        <eventuale restituzione dei risultati al cliente>;  
      }  
  od  
}
```

Primitive di comunicazione asincrone

1. Processi servitori
2. Esempi di processi servitori
3. Specifica di strategie di priorità
4. Realizzazione delle primitive asincrone

Processi servitori

a1) Risorsa condivisa che mette a disposizione di un insieme di processi clienti una sola operazione con il solo vincolo della mutua esclusione (senza condizioni di sincronizzazione).

- Soluzione in ambiente a memoria comune (monitor):

```
monitor tipo_ris
{
    tipo_var var;
    <eventuale istruzione di
    inizializzazione>
    public tipo_out fun (tipo_in x)
    {<corpo della funzione fun>;
    }
}

tipo_ris ris;
```

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

b.1) Processo servitore che offre un unico servizio senza condizioni di sincronizzazione.

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;  
  .....  
}
```

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;  
  }  
}
```

a2) Risorsa condivisa che mette a disposizione di un insieme di processi clienti due operazioni con il solo vincolo della mutua esclusione.

Soluzione in ambiente a memoria comune (monitor).

```
monitor tipo_ris{
  tipo_var var;
  {<eventuale istruzione di inizializzazione>;}
    public tipo_out fun1 (tipo_in1 x1) {
      <corpo della funzione fun1>; }
    public tipo_out fun2 (tipo_in2 x2) {
      <corpo della funzione fun2>; }
}

      ....
tipo_ris ris
```


b2) 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;
        tipo_in2 x2;
    }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; }
        }
    }
}

```

b2) Soluzione con **comandi con guardia**. Due diversi canali per i due tipi di risorse:

```
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;
    process p;
    tipo_in1  x1;  tipo_in2  x2;
    tipo_out1 y1;  tipo_out2 y2;
    {< eventuale istruzione di 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;
    od; }
}
```

a3) Risorsa condivisa che mette a disposizione di un insieme di processi clienti **due operazioni con condizioni di sincronizzazione**. (politica signal and wait).

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

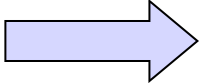
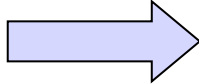
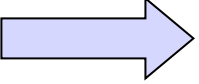
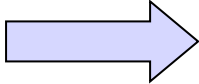
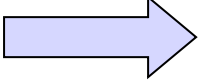
```
tipo_out2 op2 (tipo_in2 x2)
{
    .....
    if(!cond2)wait (c2);
    .....
    signal (c1);
    .....
}
```

c1,c2 variabili condizione

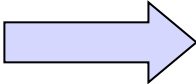
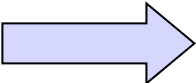
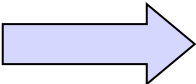
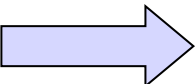
b3) Processo servitore con più servizi (due) con la specifica di condizioni di sincronizzazione

```
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;
    process p;
    tipo_in1    x1;  tipo_in2    x2;
    tipo_out1    y1;  tipo_out2    y2;
    {<eventuale istruzione di inizializzazione>;}
    do
        [] (cond1); p = receive (x1) from input1; ->
            y1=fun1(x1);
            send (y1) to p.risposta1;
        [] (cond2); p = receive (x2) from input2; ->
            y2=fun2(x2);
            send (y2) to proc.risposta2;
    od;
}
```

Corrispondenza tra monitor e processi servitori

modello a memoria comune	corrisponde	modello a scambio di messaggi
risorsa condivisa: istanza di un monitor		risorsa condivisa: struttura dati locale a un processo server
identificatore di funzione di accesso al monitor		porta del processo server
tipo dei parametri della funzione		tipo della porta
tipo del valore restituito dalla funzione		tipo della porta da cui il processo cliente riceve il risultato
per ogni funzione del monitor		un ramo (comando con guardia) dell'istruzione ripetitiva che costituisce il corpo del server

Corrispondenza tra monitor e processi servitori

modello a memoria comune	corrisponde	modello a scambio di messaggi
condizione di sincronizzazione di una funzione		espressione logica componente la guardia del ramo corrispondente alla funzione
chiamata di funzione		invio della richiesta sulla corrispondente porta del server seguito da attesa dei risultati sulla propria porta
esecuzione in mutua esclusione fra le chiamate alle funzioni del monitor		scelta di uno dei rami con guardia valida del comando ripetitivo del server
corpo della funzione		istruzione del ramo corrispondente alla funzione

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

n risorse, n processi : P_1, P_2, \dots, P_n .

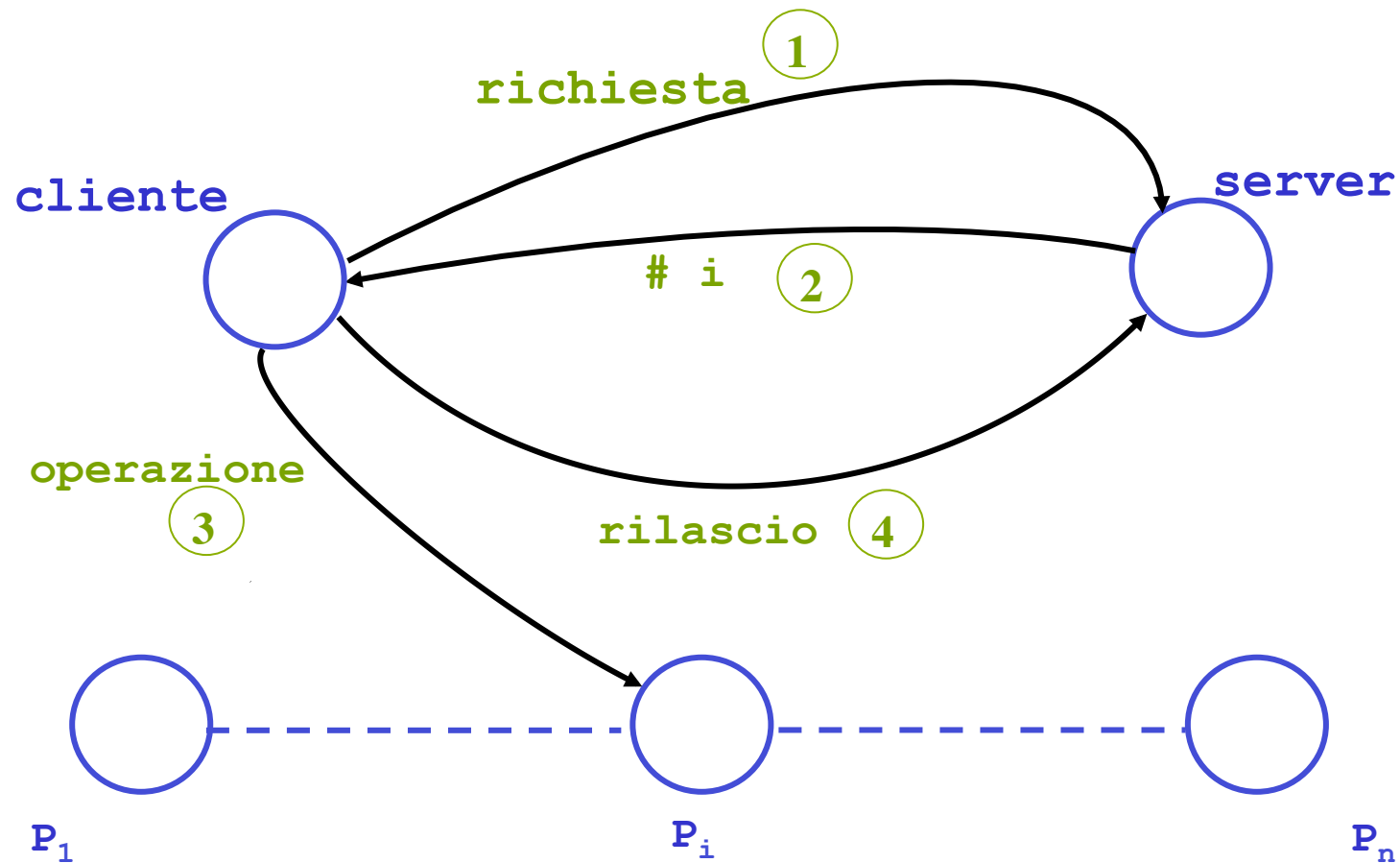
Ogni risorsa è gestita da un **processo**.

Processo **server**: indica un processo P_i attualmente disponibile.

Il **client** invia al server una richiesta per sapere 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 R_i .

Quando la risorsa non è più necessaria il client invia al server un messaggio per indicare il rilascio della risorsa

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



```

process server{
    port signal richiesta;
    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;
    od;
}

```

```

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

```

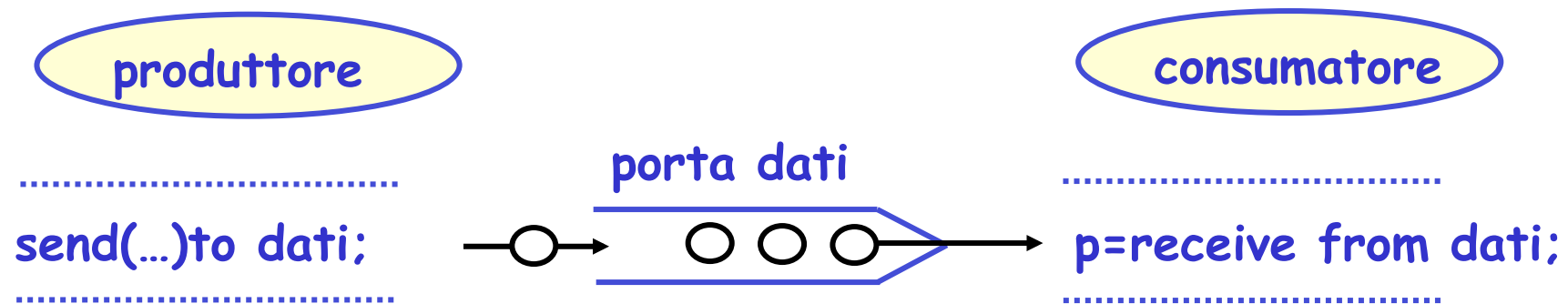
Esempi di processi servitori: simulazione di un semaforo

```
process semaphore{
    port signal P;
    port signal V;
    int  valore = 1;
    process proc ;
    signal s;
    do
        [] (valore>0) ; p=receive(s)from P; ->
            valore--;
            send (s)to proc.risposta;}
        [] p=receive (s) from V; ->
            valore++;
    od;
}
```

Cliente:

```
signal s;
/* chiamata di P:*/
send(s) to semaphore.P;
receive(s) from risposta;
...
/* chiamata di V: */
send(s) to semaphore.V;
```

**Esempi di processi servitori: scambio di dati
singolo produttore (o più produttori)-singolo
consumatore
(multi-a-uno)**



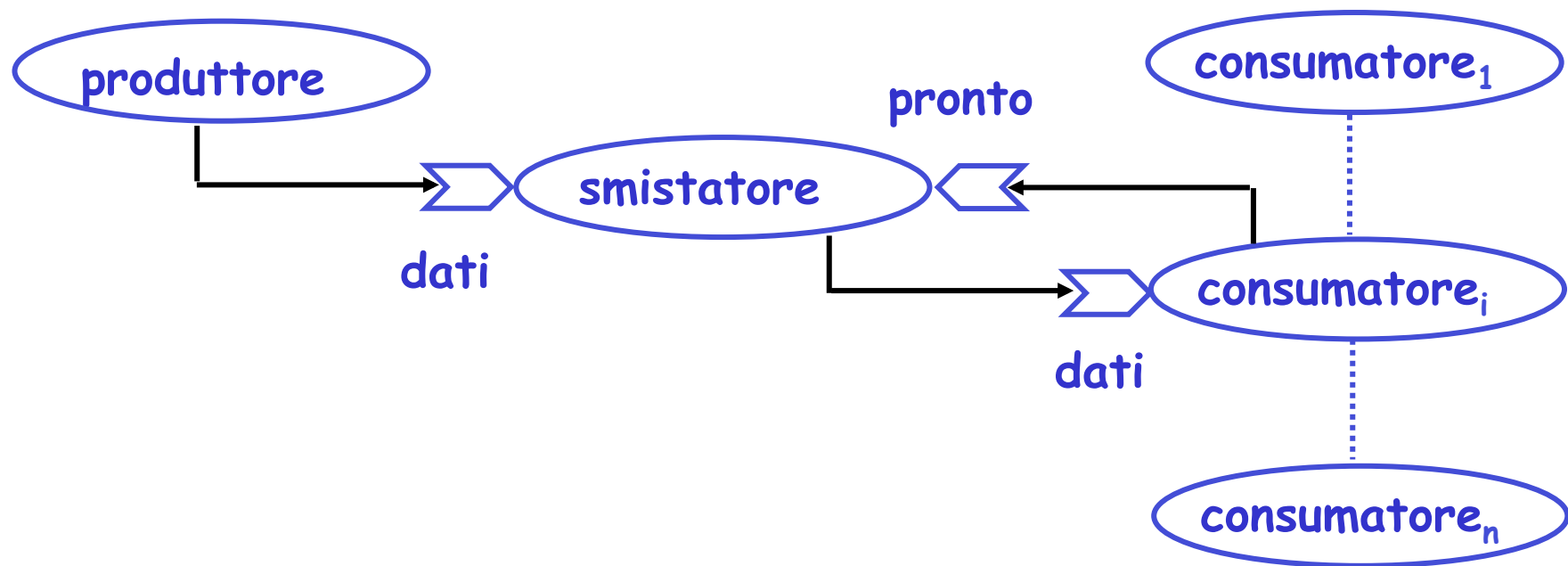
La programmazione di un processo server è superflua in quanto la soluzione è offerta direttamente dalla porta a cui un produttore invia i messaggi ed il produttore li preleva.

Nel caso esistano più processi consumatori si vogliono evitare situazioni in cui esistano processi consumatori in attesa di messaggi ed altri con messaggi in coda.

E' necessario interporre tra produttori e consumatori un processo server, a cui i produttori inviano i messaggi , con il compito di scegliere il consumatore cui inviare il messaggio in base alla disponibilità a ricevere messaggi che i singoli consumatori comunicano al server.

Le primitive send e receive eliminano la necessità di introdurre esplicitamente un buffer in quanto i canali di comunicazione contengono già, al loro interno, un buffer di dimensione praticamente illimitata.

Esempi di processi servitori: scambio di dati singolo produttore-più consumatori (uno-a-molti)



Il processo server 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 di tipo **signal**. Un messaggio di questo tipo non contiene informazioni. E' semplicemente un **segnale di controllo**.

T è il tipo generico dei messaggi inviati dai produttori. La **porta dati del server** è quindi di tipo **T**. Lo stesso vale per la **porta dati del consumatore**.

Ogni consumatore quando desidera ricevere un messaggio da un produttore, invia un **messaggio di tipo signal** 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;
    }
}
```

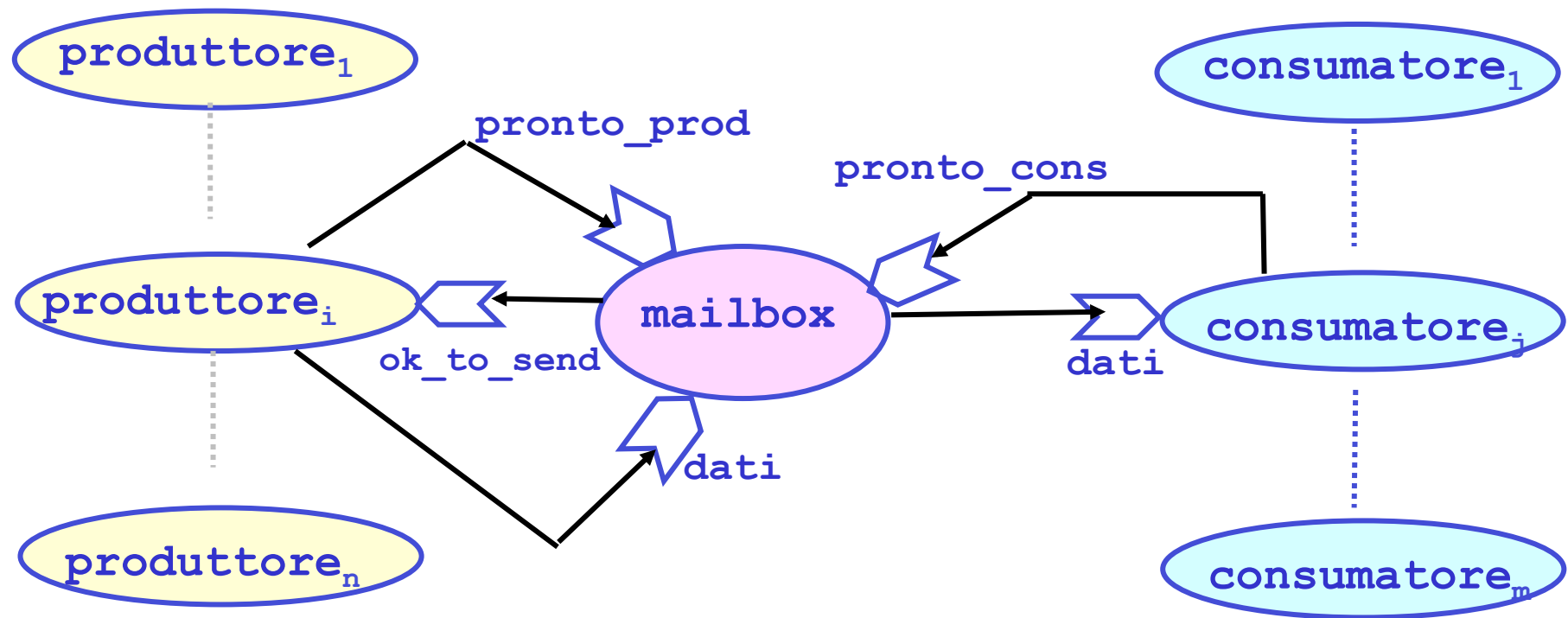
NB: la dimensione del buffer è implicitamente determinata dalla capacità del canale dati.

Esempi di servitori: mailbox (molti-a-molti)

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)



```

process mailbox{
    port T dati;
    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;
    od;
}

```

```

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

```

Specifica di strategie di priorità

Server gestisce un **pool di risorse** adottando una politica basata su priorità.

Criterio di priorità: il server deve privilegiare le richieste di P_0 rispetto a quelle di P_1 e queste rispetto a quelle di P_2

- Quando un qualunque processo P_i invia al server una richiesta , questa viene servita **se ci sono risorse disponibili**, mentre , in caso contrario, P_i 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 quelli sospesi, va scelto quello a **priorità più alta** (indice più basso).

```

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";
    }
}

```

```

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;}
od;
}

```

```

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;
  ...
}

```


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.

Le primitive di tipo sincrono possono essere tradotte dal compilatore del linguaggio in termini di primitive asincrone.

- Architetture mono e multielaboratore e architetture distribuite.

Architetture mono e multielaboratore

Ipotesi (semplificative):

1. Tutti i messaggi scambiati tra i processi sono di un **unico tipo** 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;
```

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];
    if (coda_vuota(pr->coda)) { /* sospensione*/
        blocca_su(ip);
        // context switch
    }
}

```

```

messaggio *estrai_da_porta (int ip) {
    messaggio *m;
    p_des esec = processo_in_esecuzione;
    p_porta pr = esec->portassegnazione_CPU;e_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);
}

```

```
void send (T inf, PID proc, int ip){  
    messaggio *m=new messaggio;  
    m -> informazione = inf;  
    m -> mittente = PIE(); //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: sosp
    return ris;
}
```



```
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.

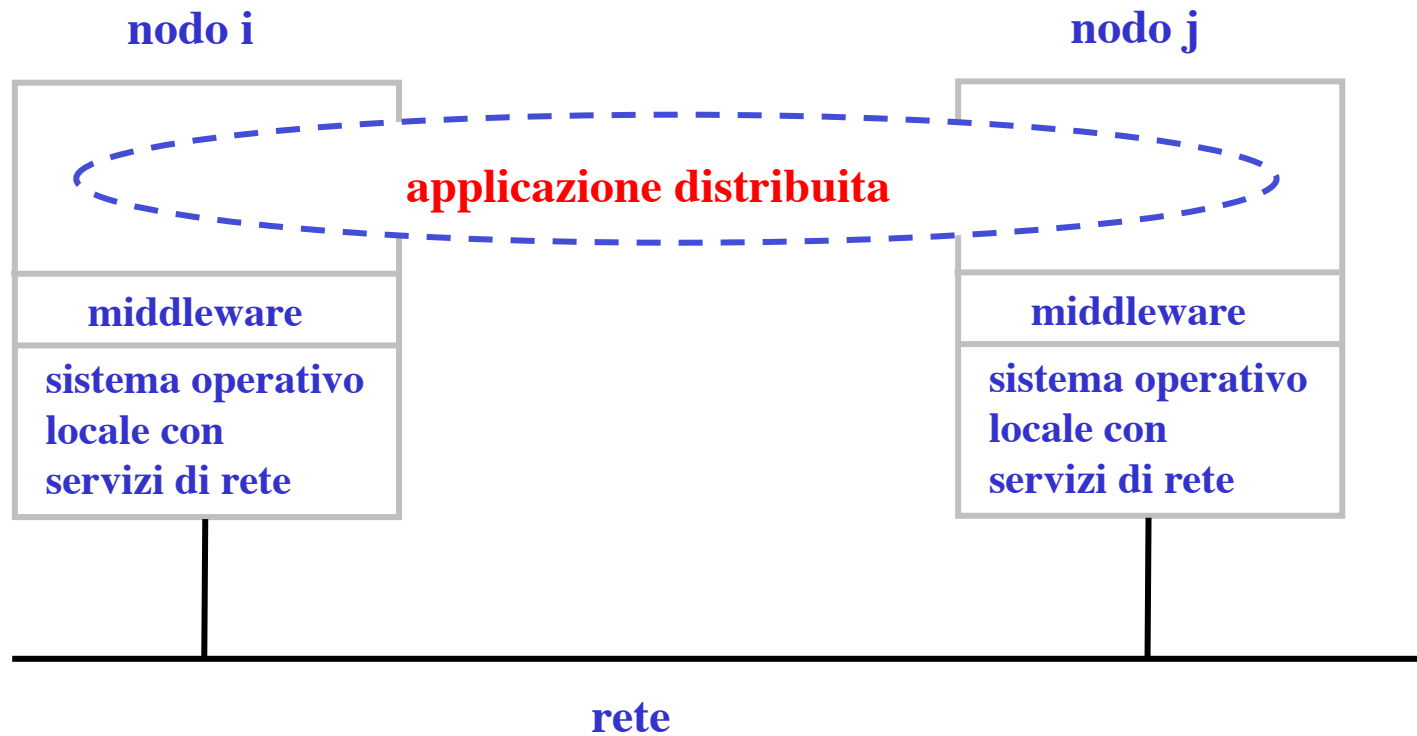
Sistemi Operativi di Rete (NOS- Network operating Systems):

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).

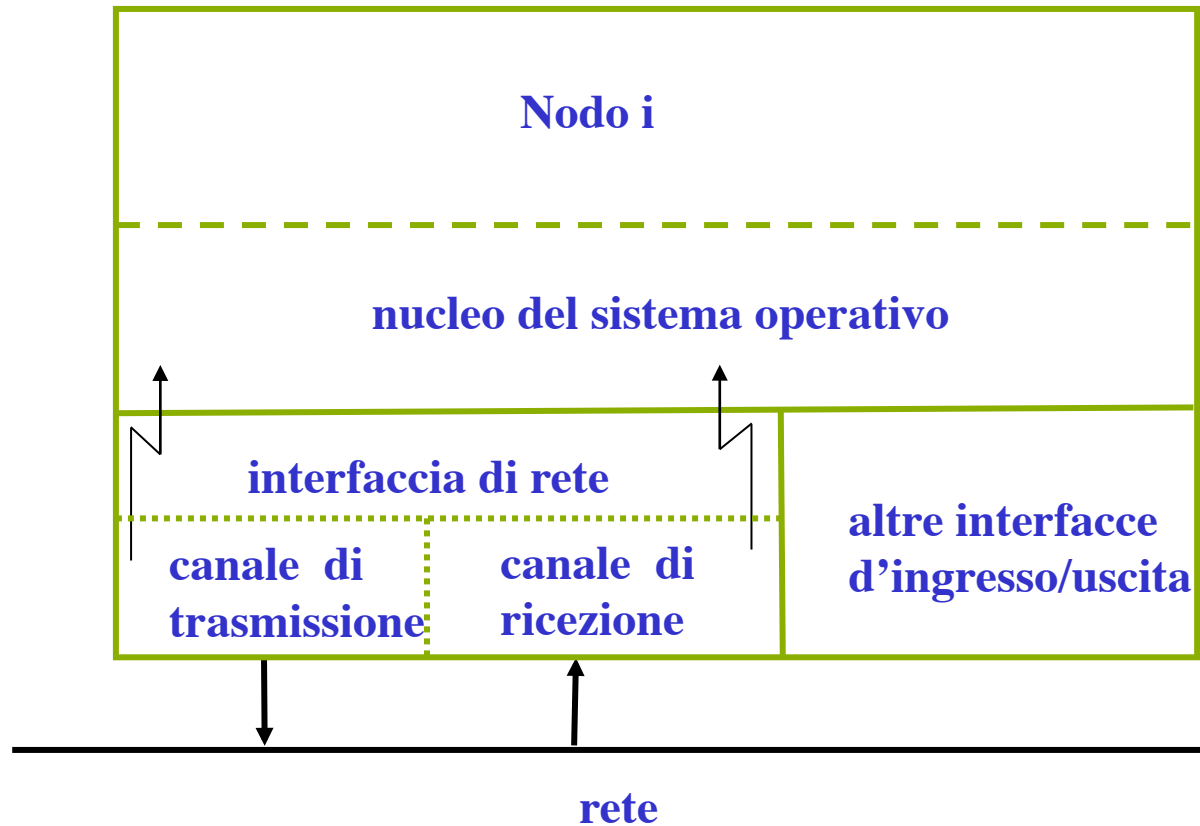
Architetture distribuite

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 **buffer** di pacchetti

Il canale di ricezione viene acceduto per realizzare la ricezione di pacchetti; ad esso è associato un **buffer** di pacchetti, nel quale vengono depositati i pacchetti inviati al nodo.

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

```

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>; }
}

```

```

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();
}

```

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.

```
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=PIE();  
    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();
}

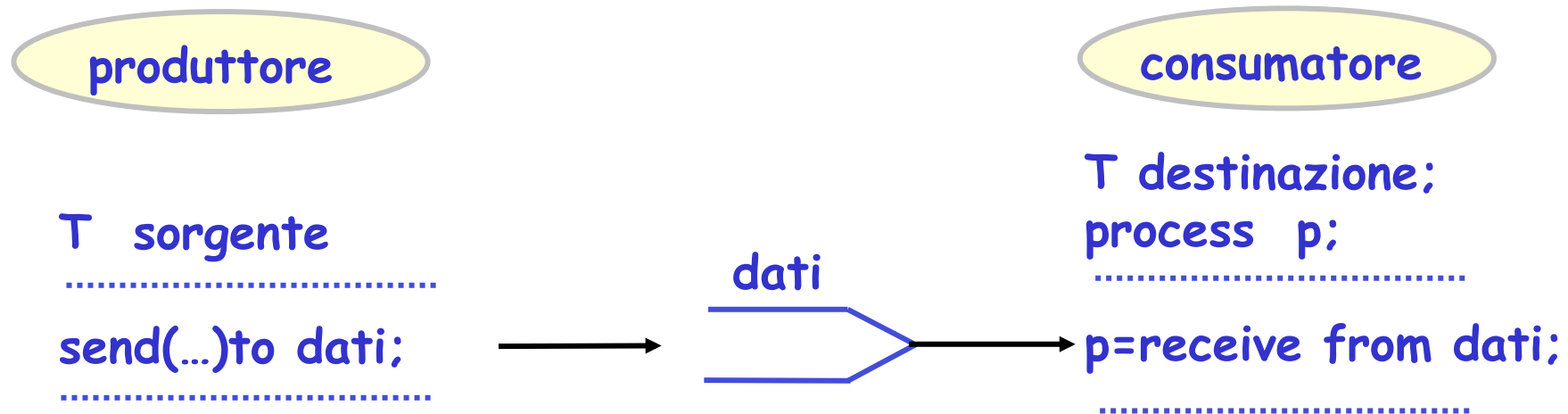
```


Primitive di comunicazione sincrone

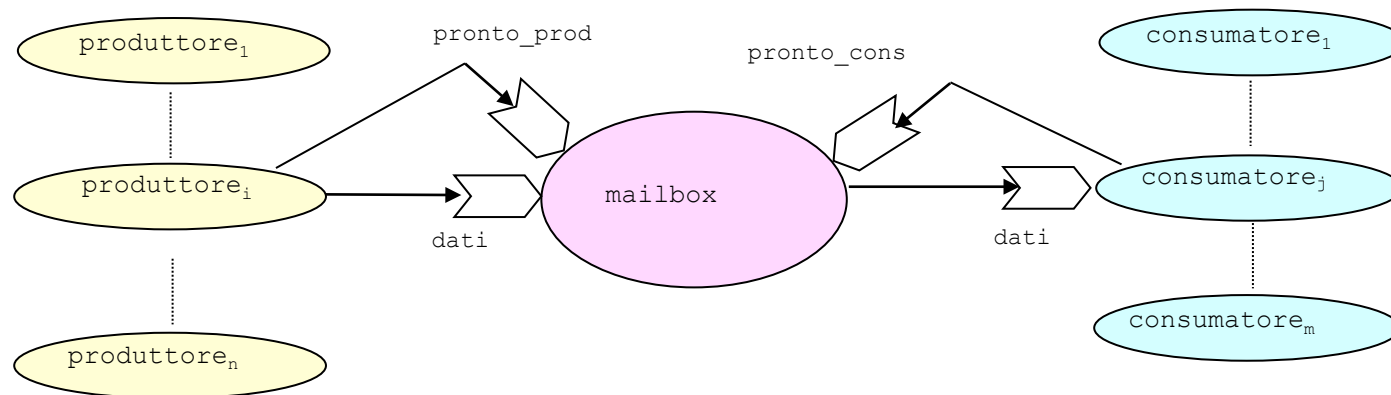
Confronto fra le primitive asincrone e le primitive sincrone

- 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  
boolean 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;
    od;
}

```

```

process produttorei{
    T messaggio;
    signal s;

    .....

    <produci il messaggio>;
    send(s) to mailbox.pronto_prod;
    send(messaggio) to mailbox.dati;

    .....

}

```

```

process consumatorej{
    port T dati;
    T messaggio;
    process p;
    signal s;

    .....

    send(s) to mailbox.pronto_cons;
    p=receive(messaggio) from dati;
    <consuma il messaggio>;

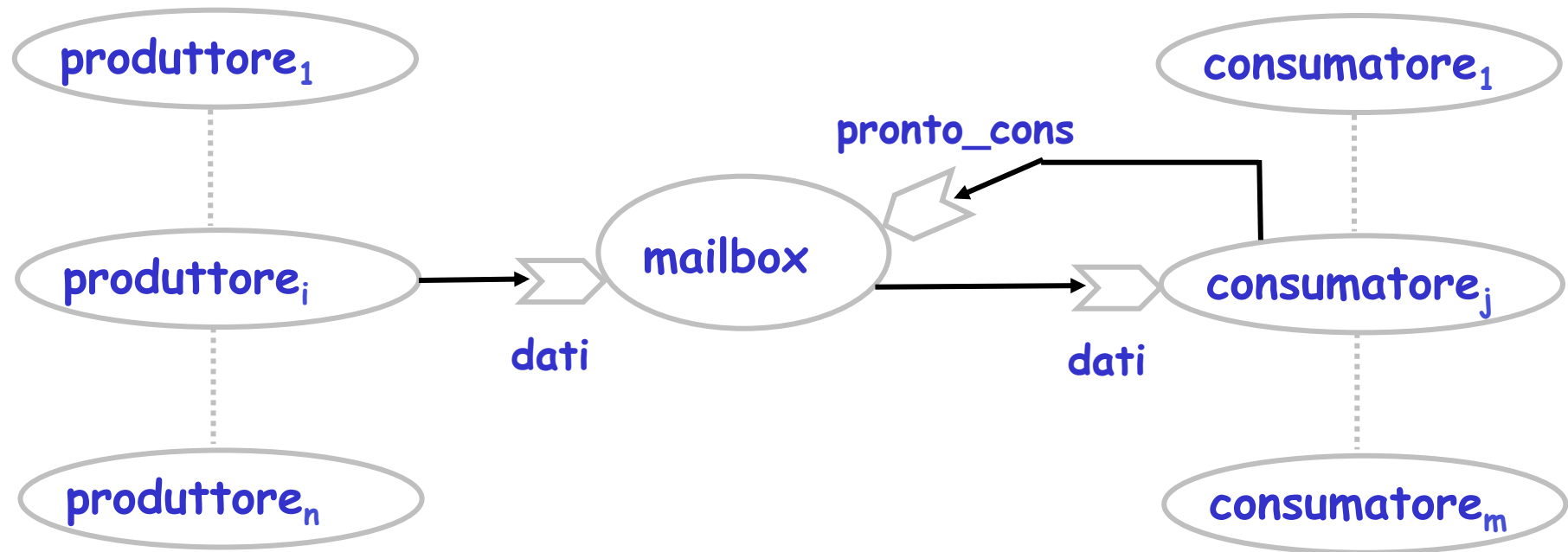
    .....

}

```

Mailbox di lunghezza finita

è possibile ottimizzare il protocollo (lato produttore):



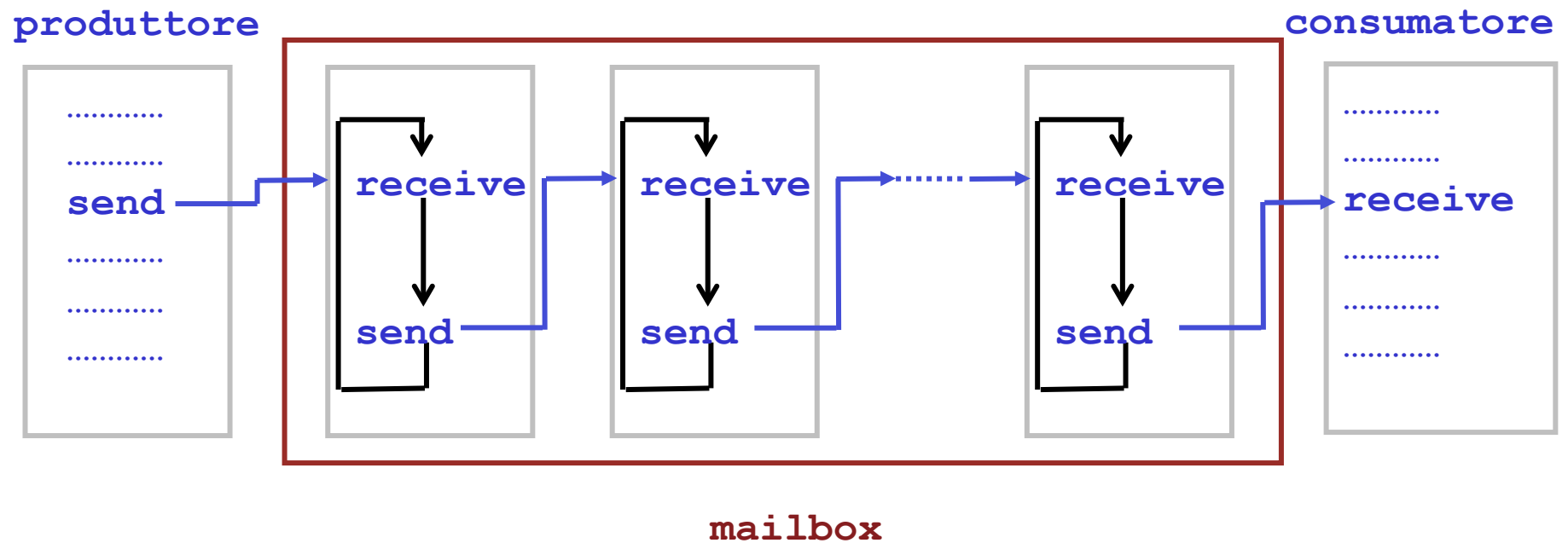

```

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;
    od;
}

```

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

Mailbox concorrente



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

Specifica di strategie di priorità (es: 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=0; 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;}

od; }

```

Realizzazione delle primitive sincrone mediante semafori

```
semaphore M=0;  
semaphore R=0;  
messaggio buffer;
```

```
public void invio(T dato){  
    messaggio mes;  
    mes.informazione = dato;  
    mes.mittente = PIE();  
    buffer=mes;  
    V(R) ;  
    P(M) ;  
}
```

```
public void ricezione(T &dato, PID &mit){  
    messaggio mes;  
    P(R) ;  
    mes=buffer;  
    V(M) ;  
    dato=mes.informazione;  
    mit=mes.mittente;  
}
```

Realizzazione delle primitive sincrone mediante primitive asincrone

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

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


Realizzazione delle primitive sincrone 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  
    .....  
}
```

```
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 = PIE();  
    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;  
}
```