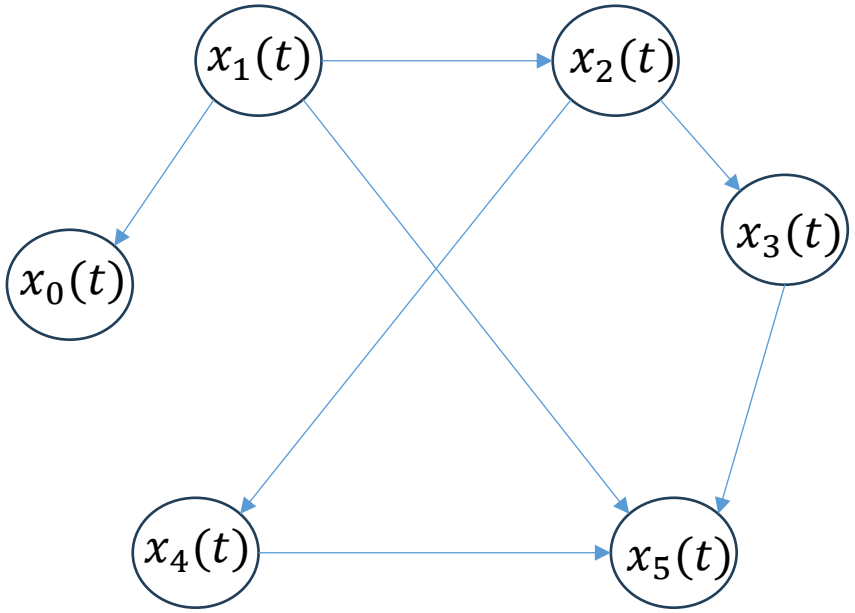
Sisteme și algoritmi distribuiți
Curs 2

Cuprins

- Modele de comunicație
- Ceasuri



Comunicație

În limbajul natural (uman) exprimarea unui mesaj cuprinde:

- Dialect
- Limbă
- Cuvînt
- Alfabet

Comunicație

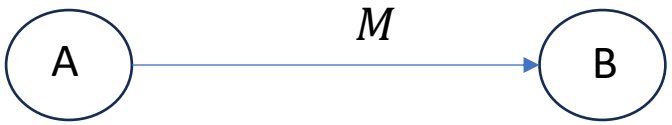
Premise necesare pentru comunicație:

- Sursa și destinația sunt conectate fizic
- Sursa alege alfabetul, limba etc., în care codifică mesajul
- Destinația decodifică mesajul (recunoaște alegerile sursei)

Modele de comunicație

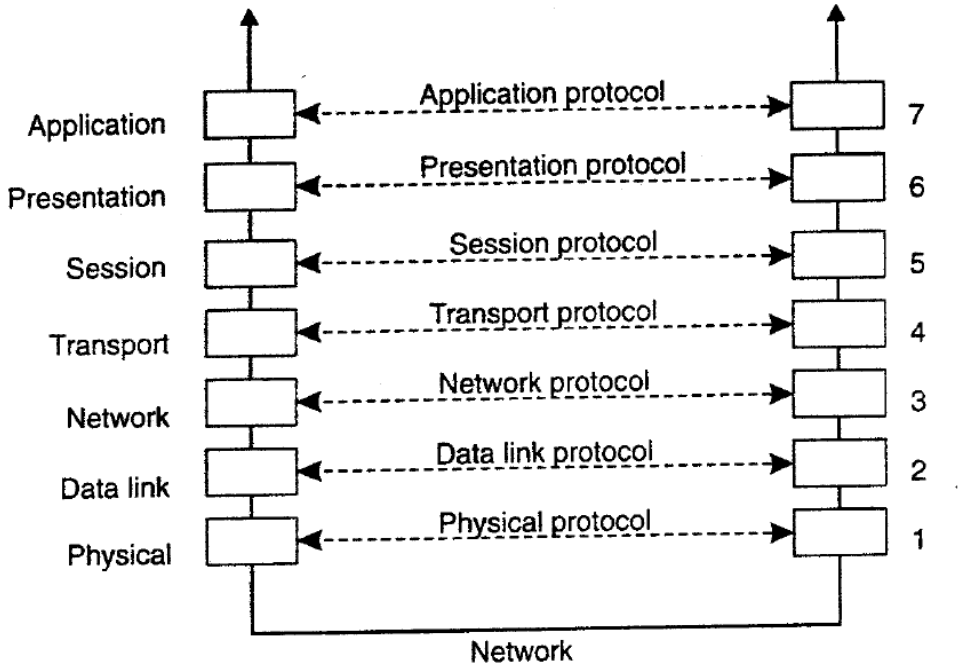
A comunică cu B:

- A produce un mesaj
- Apelează o primitivă pentru a trimite mesajul
- B apelează o primitivă pentru a primi mesajul



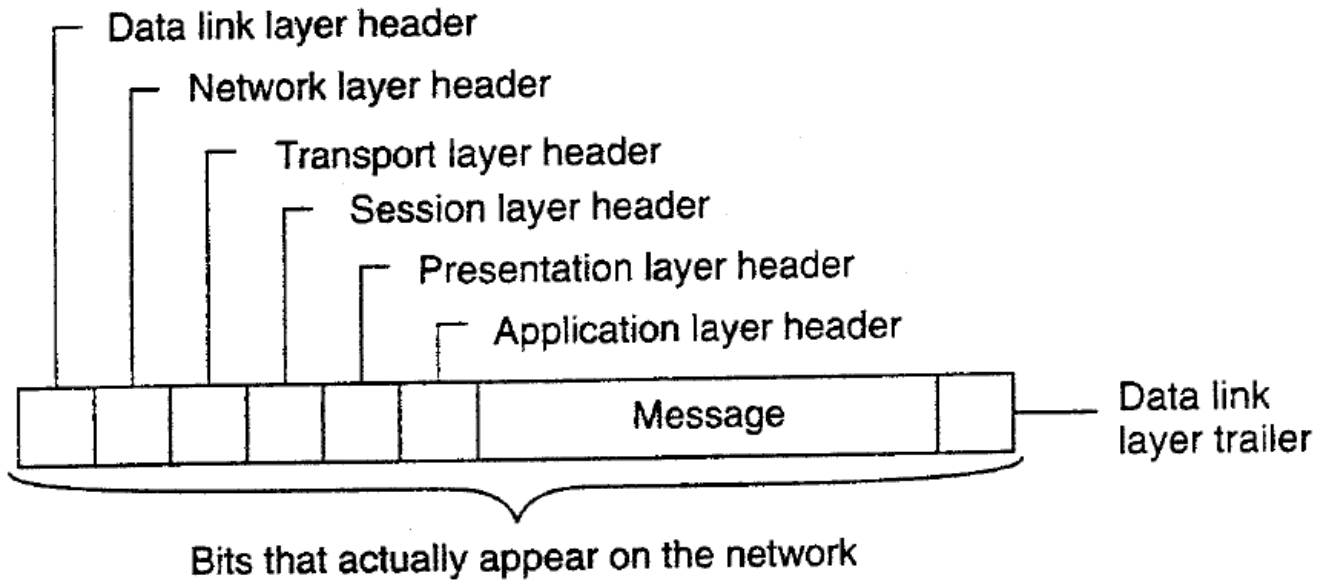
Protocoale de comunicație

Protocol = algoritm care specifică formatul, conținutul și sensul mesajelor trimise și primite de către un proces.



Protocoale de comunicație

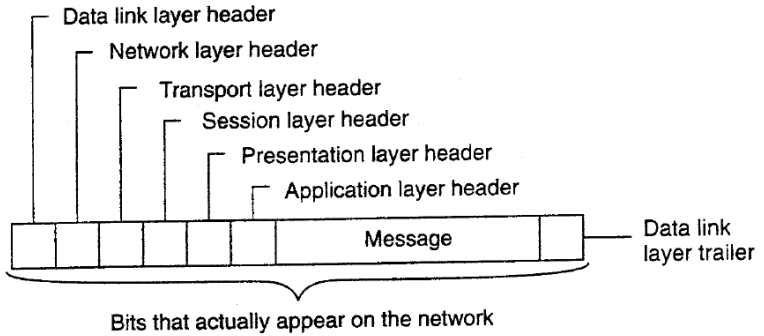
Protocol = algoritm care specifică formatul, conținutul și sensul mesajelor trimise și primite de către un proces.



Ex.: User Datagram Protocol (UDP)

Protocol de comunicație la nivelul (layer) de transport

- Fără conexiune = sursa comunică pachete către destinație fără a negocia o conexiune (implicit fără gestiunea pachetelor)
- Lightweight = nu necesită ordonare sau istoric de pachete; comunicația mesajului începe de la primul pachet
- Singurul mecanism de siguranță este *checksum*, nu există verificarea succesului transmisiei.
- Folosit în aplicațiile de streaming, broadcasting



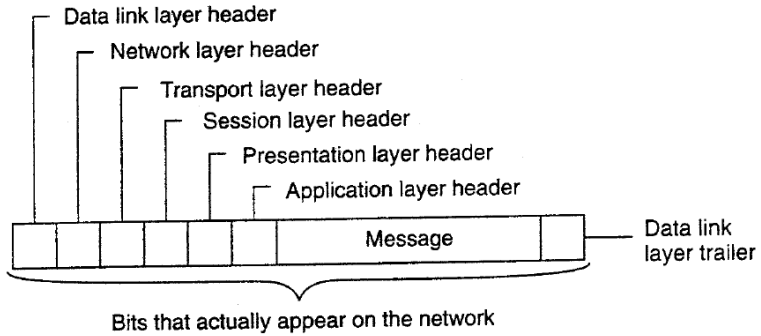
Ex.: User Datagram Protocol (UDP)

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Source Port*																*Destination Port*															
4	32	*Length*																*Checksum*															
8	64	*Data*																															
12	96																																
⋮	⋮																																

Ex.: Transmission Control Protocol (TCP)

Protocol de comunicație la nivelul (layer) de transport

- Orientat pe conexiune = înainte de interschimbarea de mesaje dintre sursă și destinație, se realizează o conexiune.
- Heavyweight = primele 3 pachete necesare pentru conexiune prin socket.
- Mecanisme de ACK, retransmisie și time-out
- Cuprinde trei etape:
  - Stabilire conexiune
  - Transfer date
  - Oprire conexiune
- Divide mesajul în porțiuni și creează segmente de dimensiuni fixe

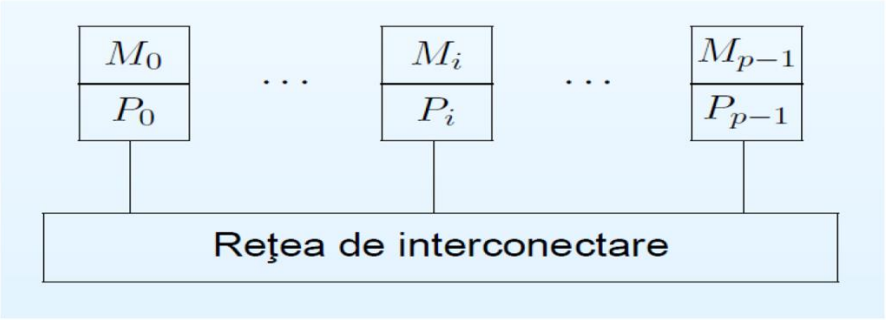


Ex.: Transmission Control Protocol (TCP)

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source Port																Destination Port															
4	32	Sequence Number																															
8	64	Acknowledgement Number (meaningful when ACK bit set)																															
12	96	Data Offset				Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window															
16	128	Checksum																Urgent Pointer (meaningful when URG bit set)[18]															
20	160	[Options] If present, Data Offset will be greater than 5. Padded with zeroes to a multiple of 32 bits, since Data Offset counts words of 4 octets.																															
⋮	⋮																																
56	448																																
60	480	Data																															
64	512																																
⋮	⋮																																

Comunicația prin mesaje: Modelul SPMD

- MIMD cu memorie distribuiă
- Paradigma **SPMD** (Single Program Multiple Data): toate procesoarele execută același program, dar fiecare utilizează un set propriu de date.
- În general, execuția programului nu este sincronă;
- Deși toate procesoarele văd același program, instrucțiunile executate nu sunt identice



Modelul SPMD

- Procesoarele sunt numerotate $0, \ldots, p$
- Numerotarea nu este statică, ci se realizează la momentul execuției.
- Există primitive care returnează adresa unui procesor (e.g. MPI_rank)
- Procesoarele pot executa instrucțiuni diferite în funcție de adresa lor

1. rank ← adresa proprie
2. **dacă** rank = 0 **atunci**
    1. a ← 1
3. **altfel**
    1. a ← 0

Modelul SPMD - variabile

- O variabilă a unui program SPMD este multiplicată (de p ori) : fiecare procesor deține un exemplar, asupra căruia are control complet.
- Un procesor nu poate modifica variabile din memoria altui procesor.
- Putem interpreta variabila a ca un vector cu p elemente: fiecare procesor P_i deține componenta i a vectorului. Cu toate acestea i reprezintă un index global al datelor din a.
- Programul inițializează a cu 0 pe toate componentele, cu excepția primei componente (care este 1).

Modelul SPMD – problemă 1

Inițializați un vector a de dimensiune n cu zero, mai puțin elementul a_m, care să fie 1.

- Cum distribuim vectorul a celor p procesoare?

- Cum facem efectiv inițializarea?

Modelul SPMD – problemă 2

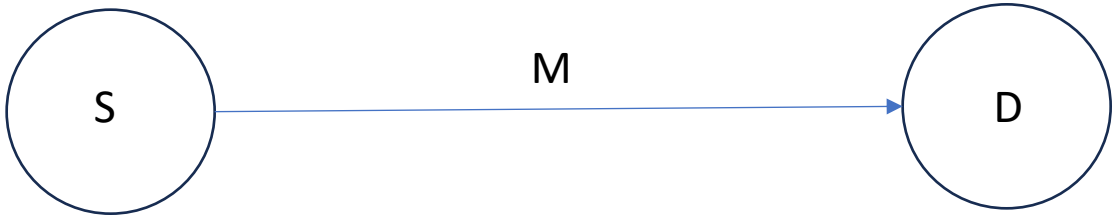
Procesorul cu id 0 deține doi vectori (reali) u,v de dimensiune n.
Realizați produsul scalar în mod distribuit.

- Cum distribuim vectorii u, v celor p procesoare?
- Ce operații de calcul intern realizează fiecare procesor?
- Cum acumulăm rezultatul?

Modelul de comunicație prin mesaje

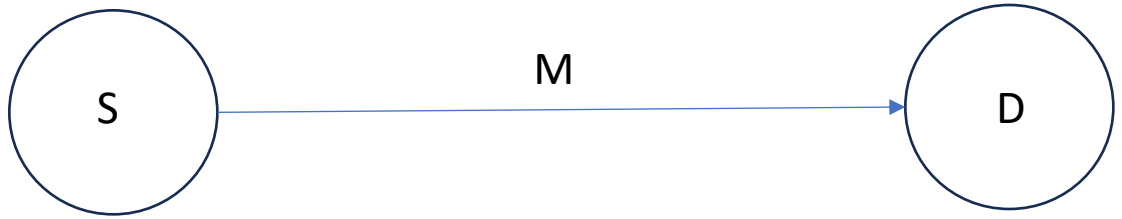
- Singura modalitate de comunicare între procesoare este transmiterea de mesaje

- **Operația de bază**:



Modelul de comunicație prin mesaje

- **Operația de bază**:

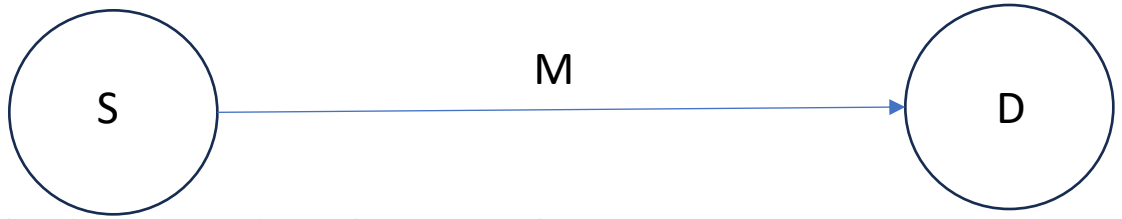


- Procesorul sursă transmite prin rutina **send**
- Procesorul destinație recepționează prin rutina **recv**

- Sintaxă generală:
  - send(date, dest)
  - recv(date, sursa)
- date = locație (buffer) din care se preiau/depun mesajele transmise
- sursa/dest = adresa procesorului cu care se comunică

MP - corectitudine

- **Operația de bază**:



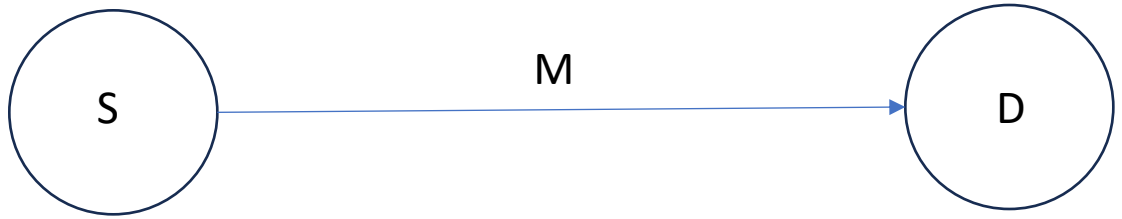
- Orice operație de **send** trebuie însoțită de una de **recv**
- Per ansamblu, vom avea perechi **send-recv**

Exemplu: Procesorul i transmite un mesaj M vecinilor de la stânga, respectiv dreapta (pe o topologie inel)

1. **dacă** rank = i **atunci**
    1. send(M, $(i+1) mod\ p$);
    2. send(M, $(i-1) mod\ p$);
2. **Altfel dacă** rank =$(i+1) mod\ p$ **atunci** recv(M, i);
3. **Altfel dacă** rank =$(i-1) mod\ p$ **atunci** recv(M, i);

MP - sincronizare

- **Operația de bază**:

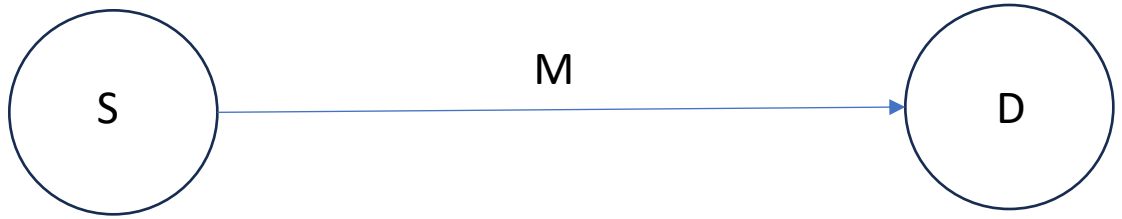


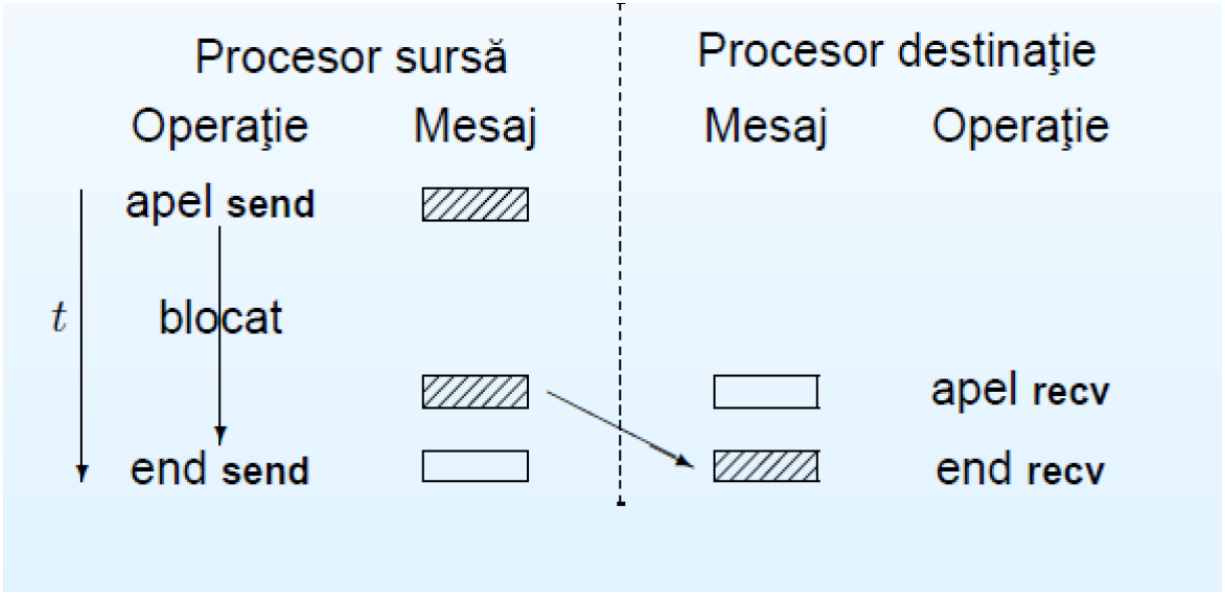
- Momentul apelului primitivei **send** este în general diferit față de momentului apelului **recv**

- Ce se întâmplă din momentul apelului primei primitive până la finalizarea comunicației?
  - Răspunsuri posibile: (i) așteaptă (**comunicație blocantă**); (ii) poate executa alte operații (**comunicație non-blocantă**)

MP – comunicație blocantă

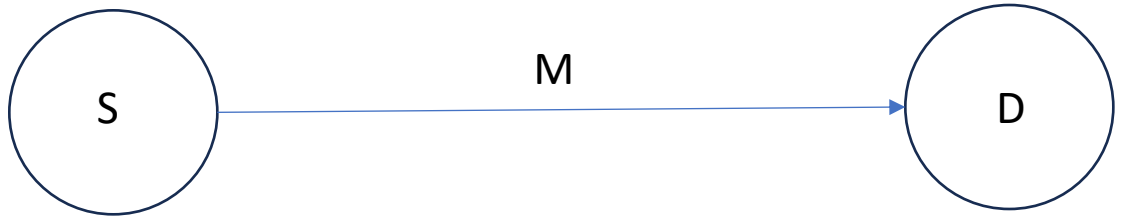
- **Operația de bază**:

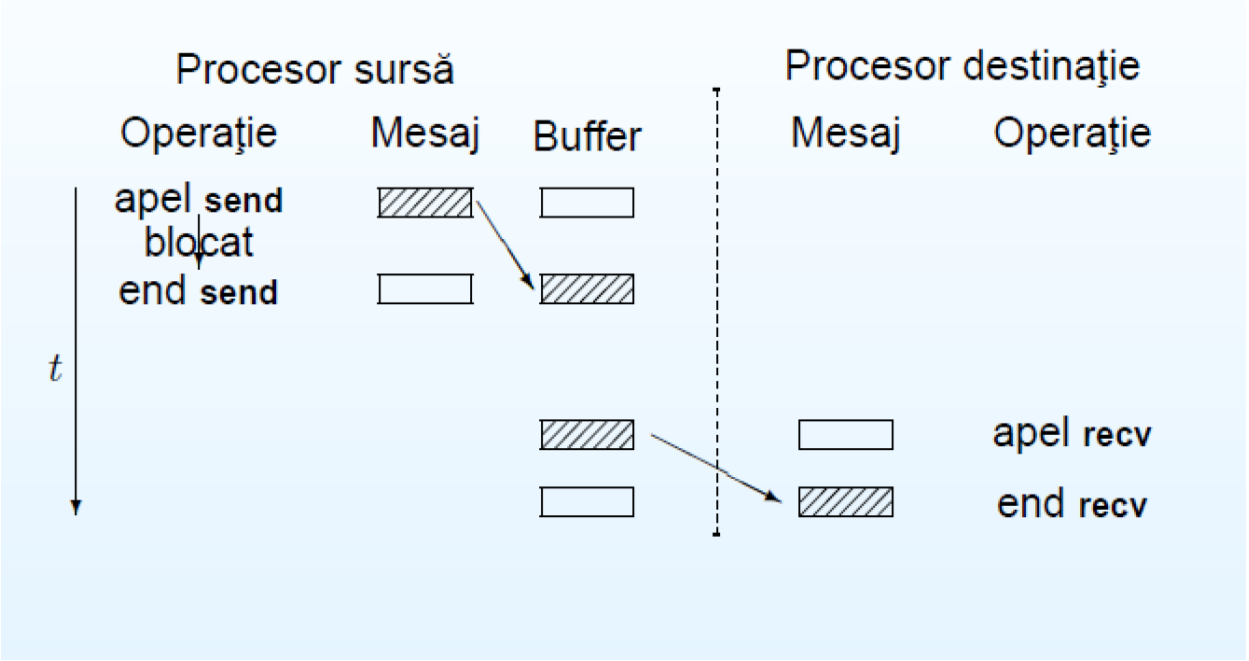




MP – comunicație blocantă prin buffer

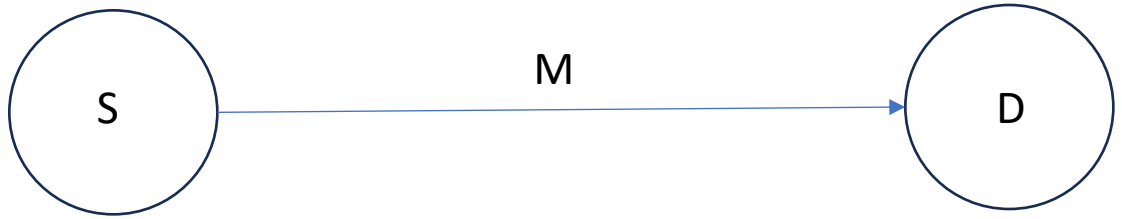
- **Operația de bază**:

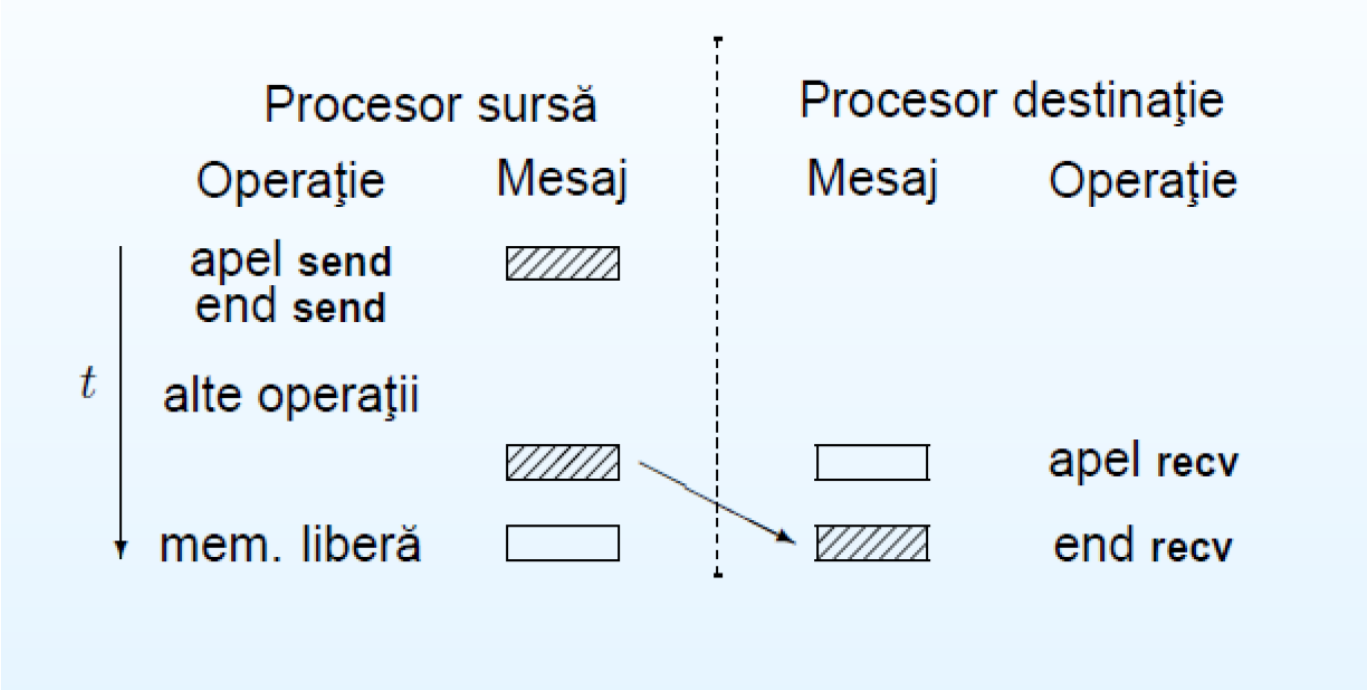




MP – comunicație non-blocantă

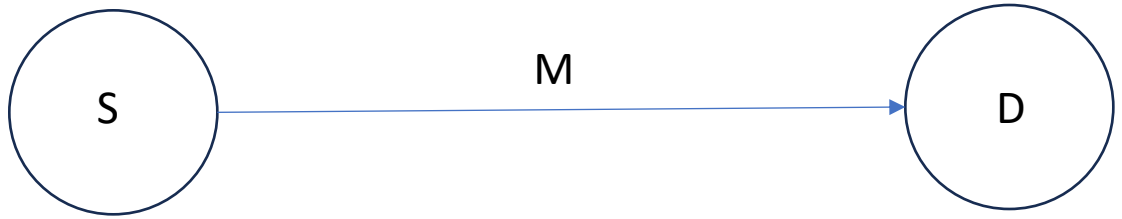
- **Operația de bază**:





MP – comunicație non-blocantă

- **Operația de bază**:



Alte primitive:

- **Așteaptă (Wait) –** așteptarea terminării comunicației

Utilizare:

1. **send**(*date*, dest);
2. execută operații care nu modifică date;
3. **așteaptă** terminarea **send**;
4. modifică date;

MP – comparație

- Comunicația non-blocantă asigură o ocupare mai bună a procesoarelor

- Erorile de programare:
  - Comm blocantă: blocarea execuției
    - send(m1, (rank + 1) mod p)
    - recv(m2, (rank - 1) mod p)

  - Comm non-blocantă: alterarea zonei de memorie după apelul **send**

Standardul MPI (Message-Passing Interface)

- Standard care descrie primitivele de comunicație în paradigma SPMD
- Implementări: OpenMPI, mpich2 etc.
- O rutină MPI se execută în cadrul unui grup de procesoare (comunicator)
- Într-un comunicator procesoarele sunt numerotate $\{0, \ldots, p-1\}$

C:

int MPI_Comm_rank(MPI_Comm com, int * my_rank)

int MPI_Comm_size(MPI_Comm com, int *p)

Python:

int comm.Get_rank()

int comm.Get_size()

Exemplu MPI

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

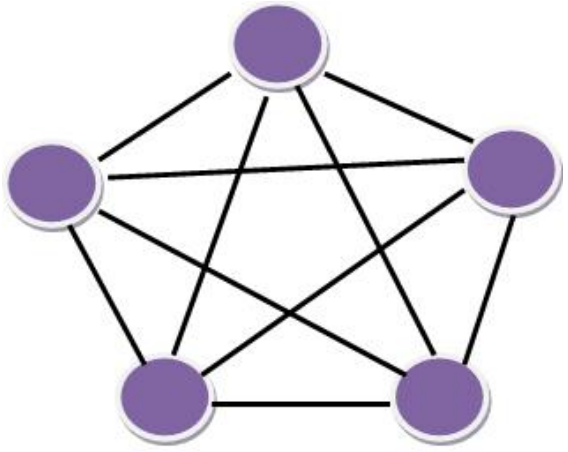
```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

MPI primitives

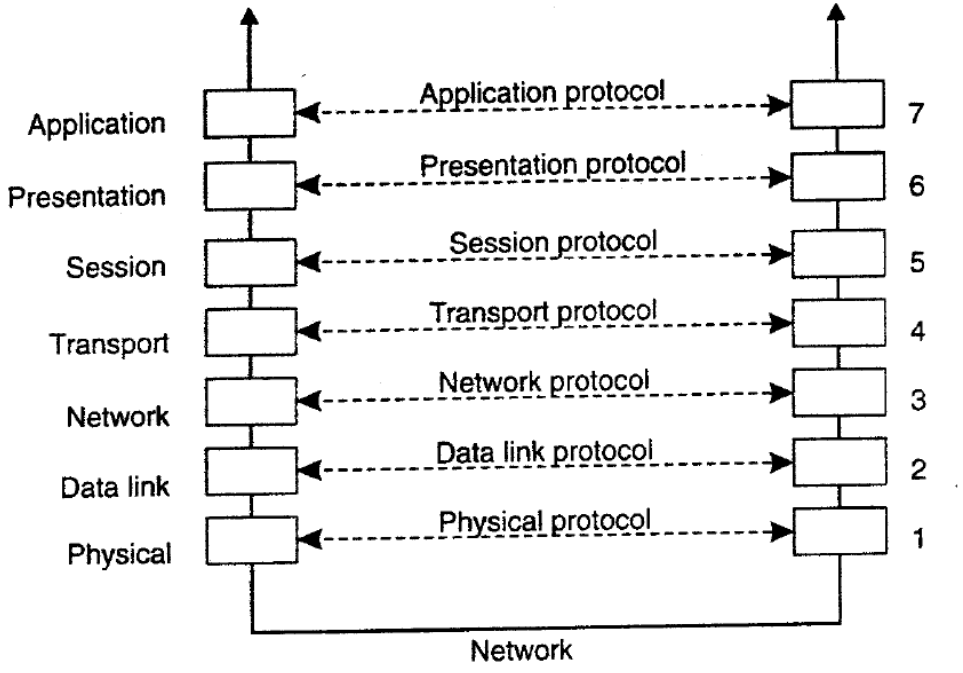
Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue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

Modele de comunicație

1. Comunicație prin mesaje
2. *Comunicație prin mesaje persistente*



http://4.bp.blogspot.com/-LLvGwl5z-_4/UMadzzwP7tI/AAAAAAAAAHY/2AxHYokPHzk/s1600/all+channel.jpg



Comunicație prin mesaje persistente (MQS)

Message Queuing Systems = sistem în care aplicațiile comunică prin inserarea de mesaje în cozi specifice;
"O abstractizare a căsuței poștale"

- Sursa (producer) are garanția că mesajul său va fi eventual inserat în coada destinatarului (receiver)
- Nu avem garanții despre momentul când (sau dacă) mesajul va fi citit
- Sursa și destinatarul execută complet independent unul de celălalt.

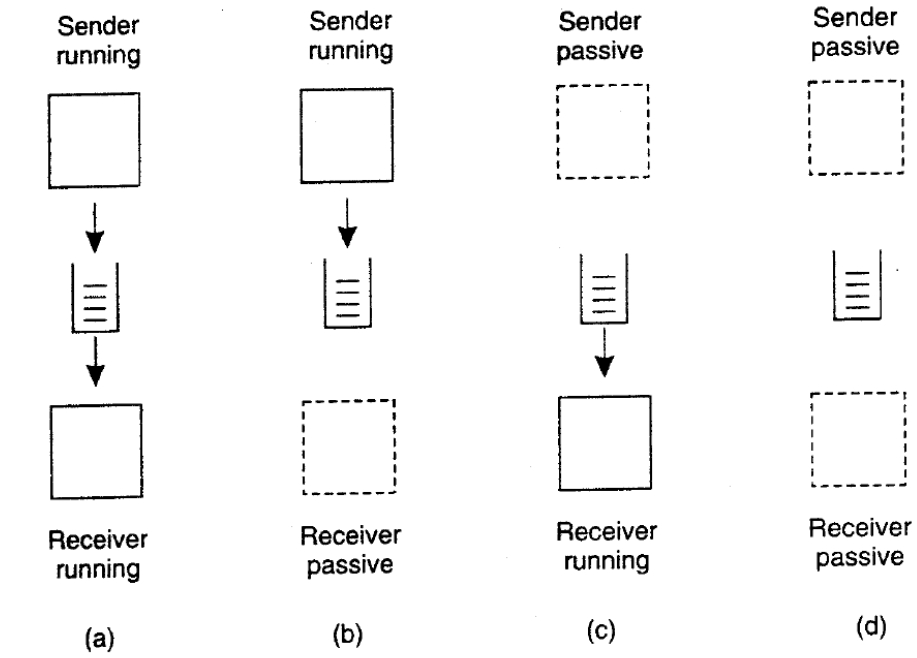


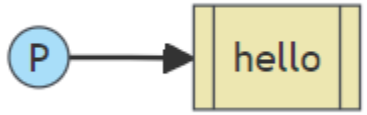
Figure 4-17. Four combinations for loosely-coupled communications using queues.

MQS primitives

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Figure 4-18. Basic interface to a queue in a message-queuing system.

Rabbit MQ example - sender




```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
```

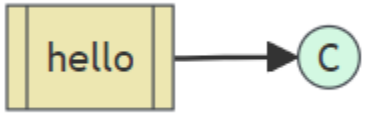
- Client RabbitMQ ⇒ modul Python pika
- Asigurăm existența cozii în care scriem mesajele (cu numele ‘hello’)

```
channel.queue_declare(queue='hello')
```

```
channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
```

```
connection.close()
```

Rabbit MQ example - receiver




```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
```

- Se realizează conexiunea identic
- Dacă nu cunoaștem a priori existența cozii create de sursă, asigurăm existența cozii în același fel

```
channel.queue_declare(queue='hello')
```

```
def callback(ch, method, properties, body):
    print(f" [x] Received {body}")
```

```
channel.basic_consume(queue='hello',
                      auto_ack=True,
                      on_message_callback=callback)
```

```
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

```
if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

Rabbit MQ example

Execuție pe receiver:

```
python receive.py
# => [*] Waiting for messages. To exit press CTRL+C
```

```
# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'Hello World!'
```

Execuție pe sender:

```
python send.py
# => [x] Sent 'Hello World!'
```

https://www.rabbitmq.com/tutorials

References

Seif Haridi, https://canvas.instructure.com/courses/902299/modules

A.S. Tanenbaum, M.V. Steen, *DISTRIBUTED SYSTEMS: Principles and Paradigms*, Pearson Prentice Hall, Second Edition, 2007.

A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, Cambridge University Press, 2008.

M. Raynal, *Distributed Algorithms for Message-Passing Systems*, Springer-Verlag, 2013.