

PPD

Clasificarea lui Flynn:

- SISD (microprocesare clasică cu arh. von Neumann)
- SIMD
- MIMD (procesare pipeline)
- MIMD (clustering)

memorie → portajată,
distribuită
hibridă

- Shared memory ⇒ 2 clase: UMA și NUMA
- cache coherent (dacă un procesor modifică o locație de memorie toate celelalte "stiu" despre această modificare)
- shared memory multiprocessors ⇒ SMP
- latență: timpul în care o date ajunge să fie disponibilă la procesor după ce s-a inițiat cererea
- bandwidth: rată de transfer a datelor din memorie către procesor

Shared Memory:

+	-
Globul address space	lipsă
Portajore date	scalabilitate
rapida și uniformă	simbol. → programator costuri mari

Arhitecturi cu memorie distribuită

+	-
Memorie scalabilă	comunicări - programatorul difícil de a mape struct. de date mari acces neuniform la memorie

massively parallel processor ⇒ MPP

- mod $\xrightarrow{\text{core local}}$

memorie fizică
spațiu de adresare
disc local și conexiuni la rețea
sistem de operare

Tipuri de arhitecturi parallele:

- Uniprocessor
 - scalar processor
 - vector processor
 - SIMD
- SMP
 - shared memory address space
 - Bus-based memory system
 - interconnection network
- Distributed memory multiprocessors
 - Message passing between nodes
 - MPP
- Cluster of SMPs
 - shared memory addressing within SMP node
 - message passing between SMP nodes
 - MPP if processor nr. is large and the communication is fast.

Proces: o instanță a unui program care se execuțiază

- { - ID
- stare
- context
- memorie

Scheduler: controlează execuția proceselor
setează stările procesului

- { new
- running
- blocked
- ready
- terminated

Thread: o parte

Race condition: condiție care poate să apară într-un sistem în care comportamentul este dependent de ordinea în care apar anumite events

Data race: un thread exec. o operatie prin care se încercă să se acceseze o locație de memorie care este în același timp accesată pt. scriere de alt thread

Lecțiune critică: segment de cod unde poate apărea data race

Instrucțiuni atomică: dacă execuția nu poate fi "interleaved" cu cea a altor instrucțiuni înainte de terminarea ei.

MPI - Structural program MPI

 MPI include file

 Declarations, prototypes

 Program begins

 Serial code

 Initialize MPI environment (Parallel code begins)

 Do work and make message passing calls

 Terminate MPI environment (Parallel code ends)

 Serial code

 Program ends

```
#include "mpi.h"
#include <stdio.h>
int main (int argc, char *argv[])
{
    int numtasks, rank, rc;
    rc = MPI_Init (&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf ("Error starting MPI program... ");
        MPI_Abort (MPI_COMM_WORLD, rc);
    }
    MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    printf ("Nr. tasks = %d My rank = %d\n", numtasks, rank);
    /* do work */
    MPI_Finalize ();
}
```

MPI_Send (buffer, count, type, dest, tag, comm)

MPI_Recv (buffer, count, type, source, tag, comm, status)

MPI_Barrier (comm, ierr)

MPI_Bcast (&msg, count, MPI_INT, source, MPI_COMM_WORLD)

MPI_Scatter (sendbuf, sendcount, MPI_INT, recvbuf, recvcount, MPI_INT, root, MPI_COMM_WORLD)

MPI_Gather (= ==)

MPI_Reduce (sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dest, comm)

Interacțiunile între procese/threaduri

comunicare
sincronizare

Sincronizare

- excludere mutuală (se evită utilizarea simultană de către mai multe procese a unei resurse critice)
- pe condiție (se amână execuția unui proces pînă când o anumită condiție devine TRUE)

Deadlock: un grup de procese/threaduri se blochează la infinit pentru că fiecare proces așteaptă după o resursă care este reținută de alt proces care la rândul lui așteaptă după alta resursă.

Starvation: dacă unui thread nu i se aloca timp de execuție CPU time pt. că alte threaduri folosesc CPU Thread este "starved to death" pt. că alte threaduri au acces la CPU în locul lui.
"Fairness" - toate threadurile au rante egale la CPU

Livelock: situația în care un grup de procese/threaduri nu progresează datorită faptului că își cedă reciproc execuția.

Operări atomici: sunt efectuate ca o singură unitate - indivizibil sau parțial, ori complet ori deloc

Semafor: primitivă de sincronizare de nivel înalt (nu cel mai) și operații indivizibile: P(S) (apelată de proces care dorește să acceseze reg. critică)

V(S) (apelată la sf. sec. critică și semnifică eliberarea acesteia pt. alte procese)

• variabilă \rightarrow count = v(S) (salvare semafor)

Dass SEMAPHORE feature

count: INTEGER

down

do areăt count > 0
count := count - 1

end

up do count := count + 1
end

end

Fără o evidență a proceselor care așteaptă intrarea în secțiunea critică
 \Rightarrow nu se poate asigura starvation-free

Weak Semaphore

{v(s), c(s)}

- valoarea semaforului
- multime de asteptare la semafor
- + op. P(s)/down și V(s)/up

Strong Semaphore

{v(s), c(s)}

- valoarea semaforului
- coada de asteptare, continde ref la procesele care asteapta la semaforul s (FIFO)
- + op P/down și V/up

count : INTEGER

blocked : CONTAINER

down

```
do
    if count > 0 then
        count := count - 1
```

```
else
    blocked.add(p) // procesul curent
    p.state := blocked // block proces P
```

```
end
end
```

up

```
do
    if blocked.is_empty then
        count := count + 1
    else
```

```
        Q := blocked.remove // select some process Q
        Q.state := ready // unblock process Q
```

```
    end
end
```

weak semaphore → posibil starvation (random)

semaphore linear → mutex

JAVA: java.util.concurrent.Semaphore package

- Constructors: Semaphore(int K)

- $\text{Semaphore}(int K, boolean b)$ strong = TRUE

- Operations: acquire() (down) throws InterruptedException

- release() (up)

desavantaje

- nu se poate det. utilizarea corecta doar din codul in care apare
- daca se pozitioneaza incorrect o operatie P sau V se compromite corectitudinea
- este ușor să se introducă deadlocks în program

- Monitor**: poate fi considerat un tip abstract de date (poate fi implementat ca și clasa) care conține din:
- set permanent de **variabile** = resurse critice
 - set de **proceduri** = operații asupra variabilelor
 - **corp de initializare** (scr. de instrucțiuni)
 - corpul e apelat la lansarea programului și produce valori initiale pt. variabilele - monitor
 - monitorul este acerat numai prin procedurile sale
 - numai una din procedurile monitorului poate fi executată la un moment dat \Rightarrow excludere mutuală
 - sincronizarea pe condiție \rightarrow variabile de tip condiție, signal (notify), wait
 - Object-oriented view
 - toate atributele sunt private
 - metodele ne exec. prin excludere mutuală

```

monitor class MONITOR_NAME
  feature
    -- attribute declarations
    a1: TYPE1
    ...
    -- routine declarations
    r1(arg1,...,argk) do ... end
    ...
    invariant
    -- monitor invariant
  end

```

```

entry: SEMAPHORE or(arg1,...,argk)
  do
    init. v(entry)=1
    entry.down
    body
    entry.up
  end

```

- Variabile conditionale**: o abstractizare care permite sincronizarea conditională
- asociate cu lacătul unui monitor
 - permit thread-urilor să aștepte în interiorul unei secțiuni critice eliberând lacătul monitorului

- Variabilă conditională**: constă dintr-o coadă de blocare și 3 operații atomice:
- wait (elib. lacătul monitorului, blochează threadul care se executa și-l adaugă în coadă)
 - signal (dacă coada este empty n-a efect altfel delocksază un thread din coadă)
 - is_empty (TRUE / FALSE, coada e empty?)
 - wait + signal \Rightarrow din corpul unei rutine a mon.

```

class CONDITION_VARIABLE
feature
    blocked : QUEUE
    wait
        do
            entry_up // release the lock on the monitor
            blocked.add(p) // p - the current process
            p.state := blocked // block p
        end
    signal deferred end // behavior depends on signaling discipline
    is_empty : BOOLEAN
        do
            result := blocked.is_empty
        end
    end

```

JAVA: fiecare obiect are un monitor care poate fi blocat sau deblocat în blocuri sincronizate
 variabilele conditionale nu sunt explicit disponibile, dar metodele {wait(), notify(), notifyAll()} pot fi apelate dintr-o bloc synchronised
 nu sunt mai multe var.cond. asociate unui monitor
 disciplina = 'Signal and Continue'
 monitoarele nu sunt starvation-free

avantaje: abordare structurată
 separation of concerns

probleme: trade-off \Rightarrow raport pt. programator si performanta
 discipline de semnalizare - risc de confuzie
 nested monitor calls

Forme de sincronizare JAVA: Synchronized Static Methods

- Nonblocking Counter - Atomic variables
- Lock
- Semaphore
- Exchanger
- Class SynchronousQueues
- ReadWriteLock
- CyclicBarrier

Call by future:

- non-deterministic: valoarea se va calcula cândva între momentul creării variabilei future și momentul când aceasta se va folosi
- eager evaluation: imediat ce future a fost creată
- lazy evaluation: doar atunci când e folosită

Future and Promise:

- the two sides of an asynchronous operation
- a caller of an asynchronous task will get a Future as a handle to the computation's result

C++11

- future } - promise
 - async
 - packaged_task
- std::packaged_task object = wraps a callable object
- **async** = execută o funcție f asincron posibil în alt thread și returnează un obiect std::future care să conțină rezultatul
 - depinde de implementare dacă std::async permite în nou thread sau dacă task-ul se va executa imediat atunci când se va crea valoarea pt. future.
- std::promise = furnizări un mecanism de a stoca o valoare sau o excepție care va fi apoi obținută din nou în obiect std::future care a fost creat prin obiectul promise
 - acțiuni:
 - make_ready
 - release
 - abandon

JAVA

- task (Runnable vs. Callable)
- Future
- Executor
- CompletableFuture
- task = activitate independentă, nu depinde de stocarea rezultatului ori 'side effects' ale altor task-urilor
- Executor = mecanism de decuplare a submiterii unei task de execuția lui, suport pt. monitorizarea execuției, se bazează pe 'schablonul' producător-consumator

OpenMP - struktura generală cod

```
#include <omp.h>
main() {
    int var1, var2, var3;
    Serial Code
    ...
    /* Beginning of parallel section. Form a team of threads.
       Specify variable scoping */
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        /* Parallel section executed by all threads */
        ...
        /* All threads join master thread and disband */
    }
    Resume serial code
    ...
}
```

Exemplu - PARALLEL Region

```
#include <omp.h>
main() {
    int nthreads, tid;
    omp_set_num_threads(4);
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Hello from %d\n", tid);
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Nr. threads = %d\n", nthreads);
        }
    }
    /* All threads join master thread
       and terminate */
}
```

reduction (+ : sum)
pragma omp for [clause...]
pragma omp section [clause...]
pragma omp single [clause...]
schedule (static | dynamic |
 guided [, chunk])

Matrix Multiply:

```
#pragma omp parallel for
private(j, k)
for(i=0; i<m; i++)
    for(j=0; j<m; j++)
        c[i][j] = 0;
    for(k=0; k<m; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j]
```

Resumat: # pragma omp parallel - rezune paralelu
- clause

pragma omp for - clause
- reduce

pragma omp sections - clause

pragma omp barrier

pragma omp critical

Timp de execuție: $t_p = (\text{masse } i : i \in \overline{0, p-1} : T_{\text{calcul}} + T_{\text{comunicatie}} + T_{\text{aspetare}})$

- complexitatea timp pt. un algoritm paralel care rezolvă o prob. $P(m)$ cu dim. m a datelor de intrare este o funcție T ce depinde de m , dor și de nr. de procesare p folosite.

• pasi calcul + pasi comunicare / acces la memorie

Overhead: • $T_{\text{all}} = \text{timp total}$

• $T_s = \text{timp serial}$

• $T_{\text{all}} = p T_p$ ($p = \text{nr. procesare}$)

• $T_o = p T_p - T_s$

Acelerarea ("speed-up"): raportul dintre timpul de execuție al celui mai bun algoritm serial cunoscut, executat pe un calculator monoprocesor și timpul de execuție al programului paralel echivalent, executat pe un sistem de calcul paralel.

$$\cdot S_p(m) = \frac{T_s(m)}{T_p(m)} \quad - m \text{ dim. datelor de intrare}$$

- p nr. procesare folosite

Eficiență: parametru care măsoară gradul de folosire a procesorilor

$$\cdot E = S_p/p$$

Legea lui Amdahl: accelerarea procesorii depinde de raportul partii secerentiale față de cea paralelizabilă

$$\cdot \text{reg (e.g. } 20\% \Rightarrow \text{reg} = 20/100)$$

$$\text{par (e.g. } 80\% \Rightarrow \text{par} = 80/100)$$

$$\text{calculul serial } T_s = \text{reg} + \text{par} = 1 \text{ unitate}$$

$$\boxed{\text{Speedup} = 1 / (\text{reg} + \text{par}/p)}$$

$$\text{par} = (1 - \text{reg}), \quad p \# \text{procesare}$$

$$p \rightarrow \infty \Rightarrow S \sim 1/\text{reg} \quad (\text{e.g. } S \sim 100/20 = 5)$$

limita superioară a accelerării este data de fractia partii secerentiale

Legea lui Gustafson: atunci când dimensiunea problemei crește, partea serială se micșorează în procent

• m - dim. prob., p # procesare

$$\text{reg}(m) = \text{fractia calculului secerental}$$

$$\text{par}(m) = \text{fractia calculului paralel}$$

$$T_p = \text{reg}(m) + \text{par}(m) = 1 \quad T_s = \text{reg}(m) + p * \text{par}(m)$$

$$\boxed{\text{Speedup} = T_s/T_p = \text{reg}(m) + p * \text{par}(m)}$$

$$\begin{aligned} & \text{reg}(m) \rightarrow 0 \\ & \Rightarrow \text{acc. liniară} \end{aligned}$$

Timp de comunicatie: $t_{\text{comm}} = t_s + t_h \cdot l + t_w \cdot m$

$$\left\{ \begin{array}{l} t_s = \text{startup time} \\ t_h = \text{per-hop time} \\ t_w = \text{per-word transfer time} \end{array} \right.$$

Costul: produsul dintre timpul de executie si numarul maxim de procesare care se foloseste.

- $C_p(m) = t_p(m) \cdot p$
- o aplicatie paralela este optimă d.p.d.v. al costului dacă valoarea acuraciei este egală sau este de același ordin de mărime cu timpul celor mai bune variante sequențiale $C_p = O(t_s)$.
- aplicatia este eficientă d.p.d.v. al costului $\rightarrow C_p = O(t_s \log p)$
- costul unui sistem paralel $(p \times T_p) \rightarrow$ reflectă suma timpului pe care fiecare procesor îl petrece în rezolvarea problemei.
- un sistem paralel se numește optimal dacă costul rez. unei probl. pe un calculator paralel este asymptotic egal cu costul serial $\rightarrow E = T_s / p T_p \Rightarrow$ optimal $E = O(1)$.

Scalabilitatea: este un parametru calitativ care caracterizează atât sistemele paralele, cât și aplicațiile paralele.

- scalabilitatea aplicatiei: abilitatea unui program paralel să obțină o creștere de performanță proporțională cu nr. de procesare și dim. problemei
- scalabilitatea măsurată modul în care se schimbă (crește) performanța unui anumit algoritm în cazul în care sunt folosite mai multe elem. de procesare.
- scalabilitatea unui sistem paralel: (softvare + arhitectură) este o măsură a capacitatii de a livra o accelerare cu o creștere liniară în funcție de nr. de procesare folosite.
- metriki pentru scalabilitate:
 - funcția de isoeficiență
 - eficiență
 - fractia serială

- Granularitatea:** este un parametru calitativ care caracterizează atât sistemele parallele, cât și aplicațiile parallele.
- granularitatea **aplicării**: dimensiunea minimă a unei unități secerentiale dintr-un program, exprimată în nr. de instrucțiuni
 - **unitate secerentială**: parte din program în care nu au loc operații de sincronizare sau comunicare cu alte proceșe.
 - granularitatea unui algoritm poate fi aproximată ca fiind raportul dintre timpul total de calcul și timpul total de comunicare
 - granularitatea **sistemului**: pt. un sistem paralel, există o valoare minimă a granularității aplicării sub care performanța scade semnificativ. Această valoare de prag este cunoscută ca și granularitatea sistemului respectiv.
 - de dorit: un calculator paralel să aibă granul mică, a.i. să poată executa eficient o gamă largă de programe \Rightarrow programele parallele să fie caracterizate de o granul mare, a.i. să poată fi execuțiate eficient de o diversitate de sisteme
 - nr. de taskuri în care o problemă se decompune determină granularitatea:

nr. mare \Rightarrow **fine-grained decomposition**
 nr. mic \Rightarrow **coarse-grained decomposition**

- granularitatea este det. de nr. de taskuri care se desfășoară pt. o problemă (multe \Rightarrow granul mic)
- de multe ori, folosirea a mai putine procesoare îmbunătățește performanța sistemului paralel

- Partitionarea:** are în vedere împărțirea problemei de programare în componente care să pot fi execuțiate concurent.
- două strategii principale de partitionare:
 - decompunerea domeniului de date
 - decompunerea **funcțională**

Descompunerea domeniului de date: aplicarea atunci când domeniul datelor e mare și regulat, ideea centrală este de a divide domeniul de date, reprezentat de principalele structuri de date, în componente care pot fi manipulate independent

- distribuții de date:

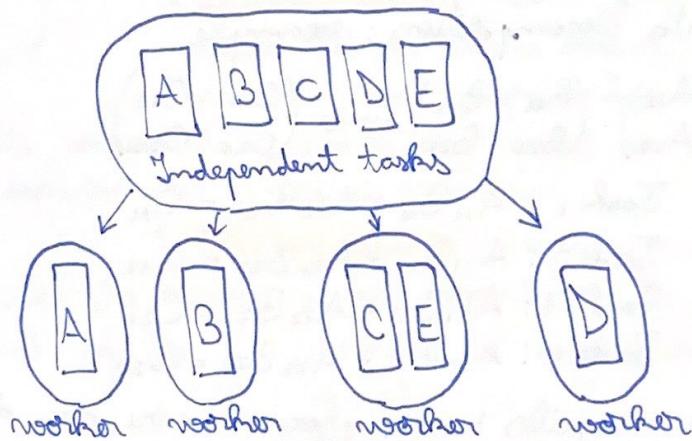
- tehnica tăierii (distribuție liniară)

- tehnica încrucișării (distribuție ciclică)

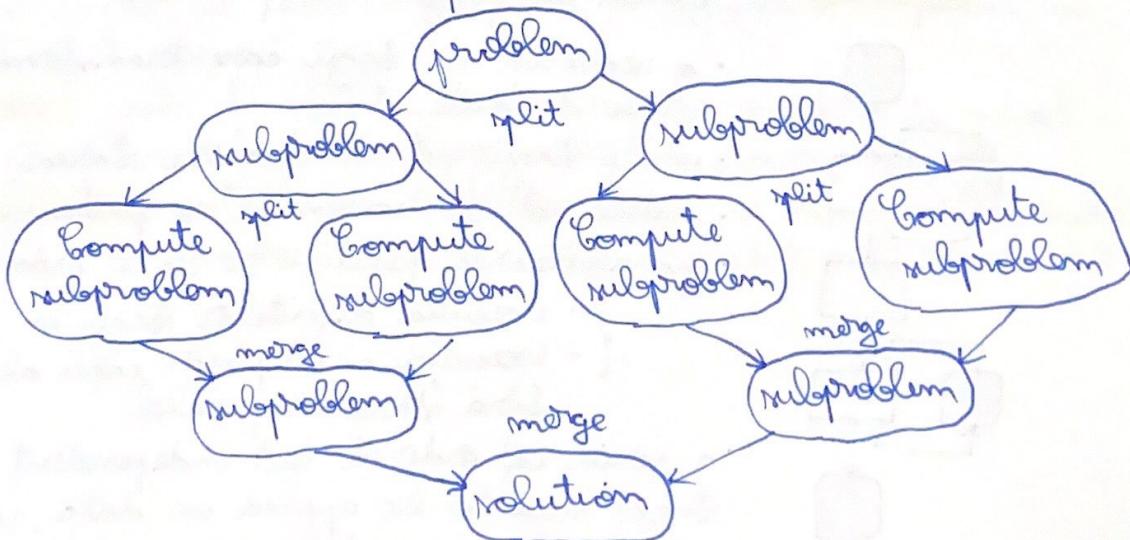
Descompunerea funcțională: tehnică de partitionare folosită atunci când aspectul dominant al problemei este funcția, algoritmul, mai degrabă decât operațiile asupra datelor.

Tâlcane de programare paralelă:

Master-Slave

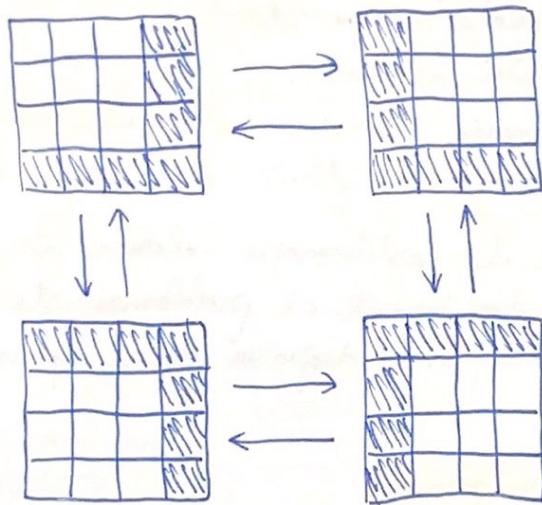


Divide et Impera



Data decomposition: descompunere geometrică

- există dependențe dar comunicarea se face într-un mod predictibil (geometric) \rightarrow vecini



Output data Decomposition: example

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \rightarrow \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\text{Task 1: } A_{11}B_{11} + A_{12}B_{21} = C_{11}$$

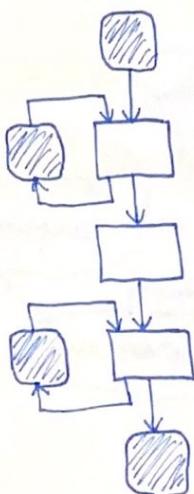
$$\text{Task 2: } A_{11}B_{12} + A_{12}B_{22} = C_{12}$$

$$\text{Task 3: } A_{21}B_{11} + A_{22}B_{21} = C_{21}$$

$$\text{Task 4: } A_{21}B_{12} + A_{22}B_{22} = C_{22}$$

- the owner computes rule: procesul care are datele atribuite lui este responsabil de calculele asociate acelei date

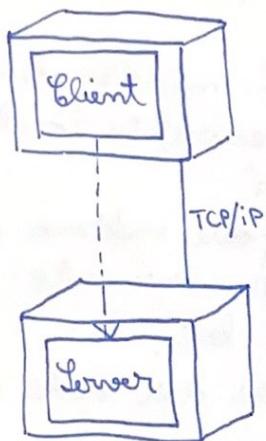
Pipeline \rightarrow răblon de programare paralelă



- o secvență de stagi împărțită care transformă un flux de date
- unul dintre stagi pot să stocheze starea
- datele pot fi "consumate" și produse incremental
- paraleлизarea pipeline se face prin:
 - execuția diferitelor stagi în paralel
 - execuția multiplelor copii ale stagiilor fără starea în paralel
- a series of ordered but independent computation stages need to be applied on data, where each output of a computation becomes input of subsequent computation.

Sistem distribuit: este format din componente hardware și software localizate într-o rețea de calculatoare care comunică și își coordonează acțiunile doar bazat pe transmitere de mesaje.

Client - Server pattern

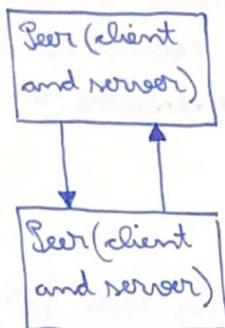


- o componentă de tip server care furnizează servicii către mai multe componente client
- o componentă client cere servicii de la componenta server
- serverele sunt active permanență 'acuțând' cererile de la clienti
- clientii și serverele lucrează în sesiuni
 - * **stateless server** (starea unei sesiuni e gestionată de către client. Această stare este trimisă împreună cu fiecare cerere)

În aplicațiile WEB, session state poate fi stocată ca și parametrii URL, în cămpuri ascunse sau folosind cookies)

- * **stateful server** (starea unei sesiuni este menținută la nivel de server și este asociată cu ID-ul clientului)
- **Transacțiile trb. să fie**
 - atomic, să asigure coherența stării
 - izolate
 - durabile
- **fault-handling** = starea menținută la nivelul clientului implică faptul că toată informația se va pierde în cazul în care clientul exizează.
- **securitatea** poate fi afectată dacă starea se menține la nivel client pt. că informația se transmite la fiecare request.
- **scalabilitatea** poate fi redusă dacă starea se menține la nivelul serverului 'in-memory'; mulți clienti => necesar memorie excesivă.

Peer-to-peer pattern



- stablon client-server simetric:

{ - un **mod (peer)** poate funcționa ca și un client - cere servicii de la alte componente sau ca și server - furnizează servicii pentru alții.
- modul unui mod se poate schimba în mod dinamic.

- stat clientui, căt și serverele folosesc usual multithreading
- serviciile pot fi simple (e.g. stream de conectare) în locul unei cereri (request) trimise prin invocare directă
- un peer care acționează ca și server își poate informa colegei care acționează ca și client de apariția unor evenimente; clientul pot fi informați folosind de ex. un event-bus.
- performanța crește odată cu nr. de noduri dar scade atunci când sunt prea puține

- **avantaje**
 - modurile pot folosi capacitatea întregului sistem (cost individual mic și beneficiu mare obținut prin partajare)
 - overheadul de administrare este redus (retelele peer-to-peer se organizează intern)
 - scalabilitate foarte bună, rezilientă la failurile comp. individuale
 - configurația sistemului se poate schimba dinamic

- **dezavantaje**
 - nu există garanție calității serviciilor
 - nu există garanție securității