**Problema comunicației în rețea**

O particularitate a majorității programelor ce comunică în rețea este aceea că trebuie să răspundă prompt la mesaje provenind din surse diferite și într-o ordine necunoscută dinainte.

Să luăm ca și exemplu un server ssh. Serverul poate avea mai mulți clienți conectați simultan. La fiecare moment, este imposibil de prezis care dintre clienți va trimite primul o comandă. Dacă serverul execută un apel **recv()** (=apel recepționant de date) blocant de pe socket-ul unui client, serverul va fi pus în așteptare până ce acel client va trimite date. Este posibil ca utilizatorul ce comandă acel client să stea 10 minute să se gândească. Dacă în acest timp un alt client trimite o comandă, serverul nu o va putea ,,vedea“ cât timp este blocat în așteptarea datelor de la primul client. Ca urmare, datele de la al doilea client vor aștepta cel puțin 10 minute pentru a fi procesate.

Problema de mai sus poate fi soluționată în mai multe moduri:   
 Sistemele de operare de tip UNIX oferă o funcție sistem, numită **select()**, care primește o listă de identificatori de socket și, opțional, o durată, și pune procesul în așteptare până când fie există date disponibile pe vreunul dintre socket-urile aferente, fie expiră durata de timp specificată în apelul funcției **select()**.   
 O abordare complet diferită este aceea de a crea mai multe procese sau mai multe fire de execuție (thread-uri). Fiecare proces sau fir de execuție urmărește un singur client. Astfel, procesul sau firul de execuție poate executa funcția **recv()** asupra socket-ului corespunzător clientului său. În lipsa activității clienților, fiecare proces sau fir de execuție al serverului este blocat în apelul **recv()** asupra socket-ului corespunzător. În momentul în care un client trimite date, nucleul sistemului de operare trezește procesul sau firul ce execută recepționarea datelor pe socket-ul corespunzător; procesul sau firul execută prelucrările necesare după care probabil execută un nou **recv()** blocant.

Cazul unui server cu mai mulți clienți nu este singura situație în care este nevoie de a urmări simultan evenimente pe mai multe fire de execuție. Alte situații similare sunt:

* un client care trebuie să urmărească simultan acțiunile utilizatorului și mesajele sosite de la server;
* un server care poate trimite un flux de date mai mare decât capacitatea rețelei sau capacitatea de prelucrare a clientului;
* un server care trebuie să preia mesaje de la clienții conectați și, în același timp, să poată accepta clienți noi.

Un aspect important ce trebuie urmărit în proiectarea unui server concurent este servirea echitabilă și egală a clienților, independent de comportamentul acestora. Dacă un client trimite cereri mai repede decât este capabil serverul să proceseze, serverul trebuie să execute o parte din cereri, apoi să servească cereri ale celorlalți clienți, apoi să revină la primul și să mai proceseze o parte din cereri și așa mai departe. Nu este permis ca un client care inundă serverul cu cereri să acapareze întreaga putere de calcul a serverului și ceilalți clienți să aștepte la infinit.

**Elemente de limbaj și biblioteci pentru comunicare în timp real**

În principiu fiecare limbaj de programare modern, are implementată această funcționalitate.

C

* **c-socket-lib**

O bibliotecă C care face mai ușor folosirea socketurilor în programe. Biblioteca modelează simplitatea API-ului bibliotecii socket Python.

Dacă puteți folosi clasa Python, puteți crea unul la fel de ușor folosind această clasă! În prezent, biblioteca va funcționa doar pe Linux.

*Sursa:* [*https://github.com/doodleincode/c-socket-lib*](https://github.com/doodleincode/c-socket-lib)

* **The-Socket**

Această bibliotecă oferă o interfață socket ușor de utilizat. Este proiectat pentru utilizarea Windows, dar poate fi ușor portat și la Linux. Noroc și să te distrezi!

*Sursa*: <https://github.com/styczynski/The-Socket>

C#

* **AwesomeSockets**

AwesomeSockets este o bibliotecă C# care facilitează comunicarea în rețea. Suportă pe deplin atât comunicarea sincronă cât și cea asincronă. Callback-urile pentru apeluri asincrone pot fi furnizate fie ca delegat, fie ca lambda.

*Sursa*: <https://github.com/nterry/AwesomeSockets>

* **NetCoreServer**

Server și client socket asincron cu latență ultra rapidă și scăzută. Biblioteca C # .NET Core cu suport TCP, SSL, UDP, HTTP, HTTPS. Protocoalele WebSocket suportând până la 10 000 de conexiuni concurente.

*Sursa:* [*https://github.com/chronoxor/NetCoreServer*](https://github.com/chronoxor/NetCoreServer)

* **wcf-sample-server-client**

Aplicație de eșantion pentru server și client WCF bazată pe C (alternativă mai bună pentru .net la socluri TCP etc)

*Sursa:* [*https://github.com/evaldsurtans/wcf-sample-server-client*](https://github.com/evaldsurtans/wcf-sample-server-client)

JavaScript

* **WebSocket**

Protocolul WebSocket, descris în specificația RFC 6455 oferă o modalitate de a schimba date între browser și server printr-o conexiune persistentă. Datele pot fi transmise în ambele direcții sub formă de „pachete”, fără a rupe conexiunea și solicitările HTTP suplimentare.

*Sursa:* [*https://javascript.info/websocket*](https://javascript.info/websocket)

* **Socket.IO**

Socket.IO permite comunicarea în timp real, bidirecțională și bazată pe evenimente. Funcționează pe fiecare platformă, browser sau dispozitiv, concentrându-se în mod egal pe fiabilitate și viteză.

*Sursa:* [*https://socket.io*](https://socket.io/)

Java

* **Light Weight Socket Library**

LWSL este o bibliotecă de socket-uri open source realizată în Java folosind pachete JSON. Este utilizat pentru a crea aplicații de backend mici, cum ar fi serverele de verificare sau alte sarcini simple. Oricine poate contribui.

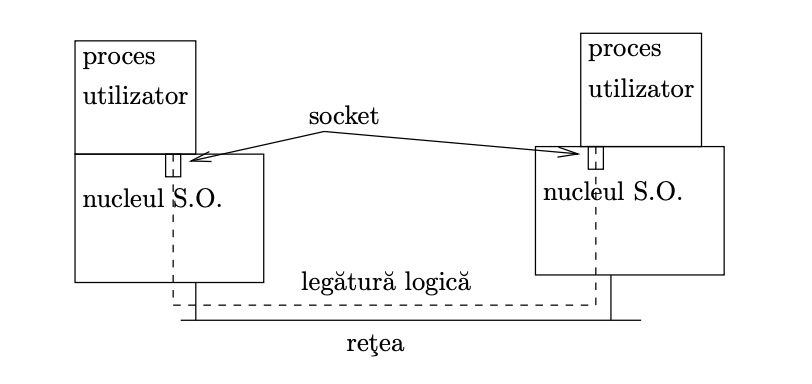
*Sursa:* [*https://github.com/StijnSimons/LWSL*](https://github.com/StijnSimons/LWSL)

* **Socket.IO-client Java**

Aceasta este biblioteca Socket.IO v1.x pentru Java, care este pur și simplu portată de la clientul JavaScript.

*Sursa:* [*https://github.com/socketio/socket.io-client-java*](https://github.com/socketio/socket.io-client-java)

**Programarea Socket-urilor**

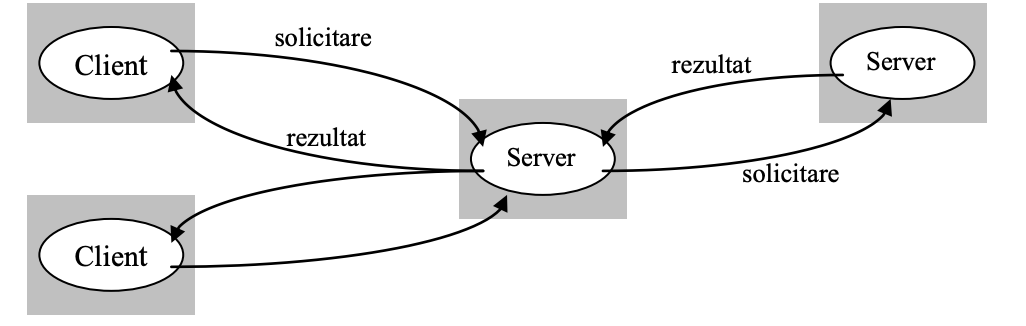
Interfața socket este un ansamblu de funcții sistem utilizate de procese pentru a comunica cu alte procese, aflate în execuție pe alte calculatoare. Interfața socket a fost dezvoltată în cadrul sistemului de operare BSD (sistem de tip UNIX). Interfața socket este disponibilă în aproape toate sistemele de operare actuale. Termenul socket se utilizează atât pentru a numi ansamblul funcțiilor sistem legate de comunicația prin rețea, cât și pentru a desemna fiecare capăt al unei conexiuni deschise în cadrul rețelei.  
  
  
 Programarea socket-ului este o modalitate de conectare a două noduri dintr-o rețea pentru a comunica între ele. Un socket (nod) ascultă un anumit port la un IP, în timp ce un alt socket ajunge la celălalt pentru a forma o conexiune.  
Principiile de bază ale interfeței socket sunt următoarele:

* Pe fiecare calculator rulează mai multe procese și fiecare proces poate avea mai multe căi de comunicație deschise. Prin urmare, pe un calculator trebuie să poată exista la un moment dat mai multe legături (conexiuni) active.
* Realizarea comunicării este intermediată de sistemele de operare de pe calculatoarele pe care rulează cele două procese. Deschiderea unei conexiuni, închiderea ei, transmiterea sau recepționarea de date pe o conexiune și configurarea parametrilor unei conexiuni se fac de către sistemul de operare, la cererea procesului. Cererile procesului se fac prin apelarea funcțiilor sistem din familia socket.
* În cadrul comunicatiei dintre procesul utilizator și sistemul de operare local (prin intermediul apelurilor din familia socket), capetele locale ale conexiunilor deschise sunt numite socket-uri și sunt identificate prin numere întregi, unice în cadrul unui proces, numite adrese. Adresa este formată conform regulilor protocolului de rețea utilizat.
* Interfața socket conține funcții pentru comunicație. Funcțiile sistem oferite permit stabilirea comunicației prin diferite protocoale (de exemplu, IPv4, IPv6, IPX), dar au aceeași sintaxă de apel independent de protocolul dorit.

Prin client desemnăm procesul care solicită în mod activ deschiderea conexiunii către un partener de comunicație specificat printr-o adresă, iar prin server înțelegem procesul care așteaptă în mod pasiv conectarea unui client.  
 Procesul client trebuie să ceară mai întâi sistemului de operare local crearea unui socket. Trebuie specificat protocolul de rețea utilizat (TCP/IPv4, TCP/IPv6, etc). Socket-ul proaspăt creat este în starea ”*neconectat*”. După crearea socket-ului, clientul cere sistemului de operare conectarea socket-ului la un anumit server, specificat prin adresa socket-ului serverului. De exemplu, pentru protocolul TCP/IPv4, adresa partenerului se specifică prin adresa IP și numărul portului. Funcțiile sistem apelate sunt: **socket()** pentru crearea socket-ului și **connect()** pentru deschiderea efectivă a conexiunii.  
 Procesul server începe tot prin a cere sistemului de operare crearea unui socket de tip conexiune pentru protocolul dorit. Acest socket nu va servi pentru conexiunea propriu-zisă cu un client, ci doar pentru așteptarea conectării clienților. După crearea acestui socket, serverul trebuie să ceară sistemului de operare stabilirea adresei la care serverul așteaptă cereri de conectare și apoi cere efectiv începerea așteptării clienților. Funcțiile apelate în această fază sunt, în ordinea apelării: **socket()** pentru crearea socketului, **bind()** pentru stabilirea adresei și **listen()** pentru începerea așteptării clienților. Preluarea efectivă a unui client conectat se face prin apelarea unei funcții sistem numită **accept()**.  
 Odată deschisă conexiunea, clientul poate trimite informație către server și vice-versa. Cele două sensuri de comunicație funcționează identic și complet independent (trimiterea datelor pe un sens nu este condiționată de recepționarea datelor pe celălalt sens). Octeții trimiși de către unul dintre procese spre celălalt sunt plasați într-o coadă, transferați prin rețea la celălalt capăt și citiți de către procesul de acolo.  
 Trimiterea datelor se face prin apelul funcției **send()** *(sau, cu funcționalitate mai redusă,* ***write()****)*. Recepționarea datelor trimise de către partenerul de comunicație se face prin intermediul apelului sistem **recv()** *(sau, cu funcționalitate mai redusă,* ***read()****).*  
 Închiderea conexiunii se face separat pentru fiecare sens și pentru fiecare capăt. Există două funcții: **shutdown()** care închide, la capătul local al conexiunii, sensul de comunicație cerut de procesul apelant; și **close()** care închide la capătul local ambele sensuri de comunicație și distruge socket-ul, eliberând resursele alocate.

**Aplicații distribuite client-server**

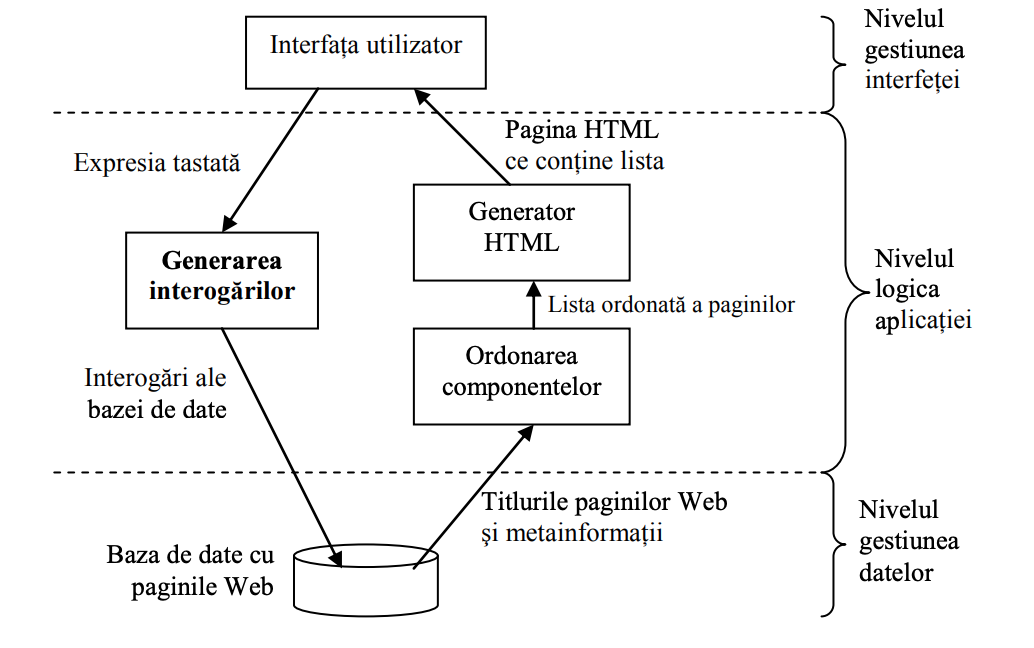
Modelul client-server este de departe cea mai cunoscută şi mai utilizată arhitectură de sistem la dezvoltarea sistemelor distribuite, fiind prezentată schematic în figura de mai jos. În fapt, ea presupune împărţirea sarcinilor aplicaţiei în procese client şi procese server care interacţionează între ele prin schimbul de mesaje în vederea realizării unei activităţi.



Ideea principală a conceptului client/server este serviciul. O aplicaţie informatică distribuită dezvoltată după modelul client/server este descompusă în două grupuri de procese: consumatorii de servicii, numiţi client şi furnizorii de servicii, numiţi server, care comunică între ele prin schimbul de mesaje de tip solicitare-răspuns. Serverul este funcţional independent de client, iar relaţia între client şi server este de colaborare (cooperare).  
 Arhitectura client/server poate fi definită ca un model de dezvoltare a aplicaţiilor conform căruia sistemul informaţional este descompus într-un mare număr de funcţii server, executate pe una sau mai multe platforme hardware, care furnizează servicii comune unui mare număr de funcţii client, executate pe una sau mai multe platforme hardware diferite dar interconectate, şi care realizează sarcini bine definite în legătură cu serviciile furnizate de server.  
 De asemenea, un server poate apela la serviciile furnizate de către un alt server, obţinându-se o relaţie client server pe mai multe straturi. O primă încercare în acest sens a constituit-o împărţirea aplicaţiilor pe două straturi, rezultând arhitectura cu două straturi. Această arhitectură presupune descompunerea aplicaţiei în: stratul corespunzător aplicaţiei și stratul corespunzător bazei de date.

Arhitectura cu trei straturi presupune împărţirea aplicaţiei în următoarele straturi: gestiunea interfeței utilizator (gestiunea prezentării), logica aplicaţiei și gestiunea datelor. În esenţă, arhitectura pe trei straturi diferă de cea pe două straturi prin separarea logicii afacerii într-un strat distinct, localizat de regulă pe un server de aplicaţii care comunică strâns cu serverul de baze de date.

Un exemplu tipic de arhitectură pe trei straturi îl reprezintă modul de funcţionare al unui motor de căutare pe Internet, prezentat în figura următoare:



Interfaţa (partea de front-end) permite utilizatorului să introducă expresia după care doreşte să se efectueze căutarea şi, ulterior, va afişa o listă cu titlurile de pagini Web care corespund expresiei introduse. Partea de back-end va consta dintr-o imensă bază de date cu informaţii despre paginile Web. Între cele două niveluri se află “inima” motorului de căutare, respectiv partea de logică a programului, care transformă expresia introdusă de utilizator prin intermediul interfeţei în una sau mai multe interogări ale bazei de date, după care va ordona rezultatele interogărilor într-o listă, pe care o va transforma într-o serie de pagini HTML.

**Aplicații paralele și concurente master-slave**

Protocolul master-slave este un model de comunicație sau control asimetric în care un dispozitiv sau proces controlează unul sau mai multe alte dispozitive sau procese și servește ca un punct de control al comunicării acestora. În unele sisteme, un master este selectat dintr-un grup de dispozitive eligibile, celelalte dispozitive acționând în rolul de slave.

Programarea paralelă este arta de a programa o colecţie de calculatoare pentru a executa eficient o singură aplicaţie. Programarea paralelă este parte componentă a programării concurente. Termenul de programare concurentă este asociat atât cu sistemele multiprocesor şi reţelele, cât şi cu sistemele de operare şi sistemele în timp real.

Conform clasificării Flynn calculatoarele paralele aparţin unuia din următoarele patru sisteme:

* SISD: sistem cu un singur set de instrucţiuni şi un singur set de date;
* SIMD: sistem cu un singur set de instrucţiuni şi mai multe seturi de date;
* MISD: sistem cu mai multe seturi de instrucţiuni şi un singur set de date;
* MIMD: cu mai multe seturi de instrucţiuni şi mai multe seturi de date.

Sistemul SIMD este conceput pe modelul master-slave datorită faptului că este compus dintr-o unitate de control (MCU - master control unit) şi un număr de procesoare identice. Unitatea de control transmite aceeaşi instrucţiune la fiecare procesor în parte. Instrucţiunile sunt executate în acelaşi timp, adică sunt sincrone. Fiecare procesor are o memorie privată şi anumite sisteme permit accesul la o memorie globală.

Sistemele SIMD sunt indicate pentru rezolvarea problemelor care pot fi descompuse în subprobleme ce presupun un efort de calcul similar (descompunere regulată). Asemenea probleme sunt:

* procesarea de imagini
* dinamica fluidelor
* automate celulare, care sunt sisteme elementare a căror comportare colectivă simulează fenomene complexe din natură.

Se disting două tipuri de sisteme SIMD: organizate pe cuvânt sau organizate pe bit. Operaţiile aritmetice într-un sistem organizat pe cuvânt sunt similare cu cele efectuate de maşinile seriale. Algoritmii pentru maşinile organizate pe bit depind de lungimea în biţi a datelor şi nu de cardinalul mulţimii de date. Mai multe instrucţiuni sunt necesare pentru a efectua o operaţie aritmetică simplă. Timpul aritmetic este mai lung decât cel dintr-un sistem organizat pe cuvânt. Creşterea vitezei nu se obţine prin creşterea vitezei fiecărui procesor în parte, ci prin utilizarea a cât mai multe procesoare simultan.

Exemplele clasice de sisteme SIMD sunt: procesoarele vectoriale și procesoarele matriceale.