# Curs 2 - Arhitecturi Distribuite: Client-Server, Multi-Tier, Peer-to-Peer

## 2.1 Arhitecturi **client-server**

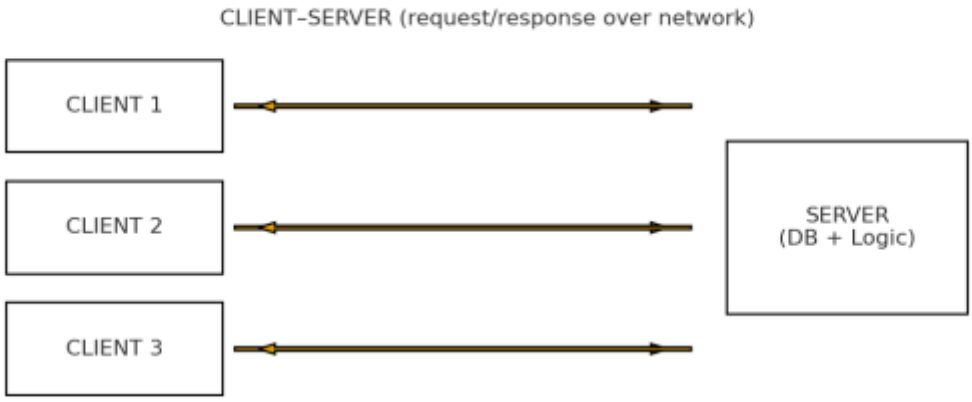


*Figura 2.1: Exemplu de arhitectură client-server tradițională – mai mulți clienți (stații de lucru) conectați la un server central printr-o rețea.*

**Arhitectura *client-server*** este un model de aplicație distribuită în care sarcinile sunt împărțite între furnizorii de servicii, numiți **servere**, și solicitanții de servicii, numiți **clienți**.
În esență, serverul este o mașină (sau un proces) centrală care **oferă resurse, date sau servicii** către unul sau mai mulți clienți.
Clienții sunt dispozitive sau aplicații care **inițiază comunicarea**, trimițând cereri (request-uri) către server, iar serverul *așteaptă* aceste cereri și răspunde furnizând serviciul sau datele solicitate[1].
Comunicația are loc de obicei printr-o rețea (ex. Internet), folosind protocoale standardizate.
În modelul client-server, **rolurile sunt bine definite**: clientul se ocupă, de regulă, de interfața cu utilizatorul (prezentarea) și inițiază acțiunile, în timp ce serverul gestionează logica aplicației, procesarea datelor și stocarea centralizată a informației[2].
De exemplu, într-o aplicație web, **clientul** poate fi browser-ul web al utilizatorului (care prezintă UI și colectează input-ul), iar **serverul** este un server web ce procesează cererile (request-urile HTTP) și interacționează cu baza de date pentru a furniza conținutul cerut. Această separare permite clienților (pot fi zeci, sute sau mii) să acceseze simultan resursele oferite de un server central.

**Mecanism de comunicare și protocoale:** Interacțiunea client-server urmează adesea modelul *request-response*. Clientul trimite o cerere către server specificând ce resurse sau operațiuni dorește, iar serverul procesează cererea și trimite înapoi un răspuns (de exemplu datele solicitate sau un mesaj de eroare, dacă nu poate realiza cererea).

Un exemplu tipic este protocolul **HTTP (HyperText Transfer Protocol)** folosit în web: clientul (browser-ul) trimite o cerere HTTP (ex. GET sau POST) către server, indicând resursa dorită și eventual parametri, iar serverul răspunde cu un cod de stare (200 succes, 404 resursă inexistentă etc.) și cu conținutul cerut (ex. o pagină HTML sau un obiect JSON). Acest ciclu cerere-răspuns se repetă ori de câte ori clientul are nevoie de ceva de la server. Comunicarea este *stateless* în cazul HTTP (fiecare cerere este independentă), însă aplicațiile pot menține stare folosind sesiuni, cookie-uri sau token-uri, după caz.

**Avantaje și caracteristici:** Modelul client-server introduce **centralizare**: serverul acționează ca un hub central care concentrează logica de afaceri și datele. Această centralizare aduce mai multe beneficii.

În primul rând, **administrarea și securitatea** sunt mai ușor de asigurat la nivel central – datele fiind stocate pe server, se pot aplica politici de backup, control acces, criptare și monitorizare unitară. De asemenea, *mentenanța* este simplificată: dacă logica aplicației trebuie modificată, de obicei se actualizează doar serverul, fără a fi nevoie să se modifice fiecare client.

Serverele bine proiectate pot gestiona eficient mai mulți clienți simultan (prin multithreading, async I/O, procesare concurentă etc.), iar **scalarea** pe orizontală este posibilă prin adăugarea de servere suplimentare în spatele unui load balancer pentru a deservi mai mulți clienți.

Totodată, centralizarea permite implementarea unor **măsuri de securitate robuste** într-un singur loc – de exemplu autentificarea și autorizarea utilizatorilor se pot face de către server pentru toate cererile, astfel încât clienții (potențial heterogeni) nu au responsabilități majore de securitate.

**Limitări și dezavantaje:** Pe de altă parte, arhitectura client-server prezintă și câteva provocări. Fiind centralizat, **serverul devine un punct unic de eșec** (*single point of failure*) - dacă serverul cade sau are o problemă, întregul serviciu devine indisponibil pentru toți clienții.

De aceea, în practică se utilizează mecanisme de redundanță: servere oglindite (clustere active-pasive pentru failover) sau distribuirea încărcării pe mai multe servere active (clustere active-active, load balancing) pentru a evita supraîncărcarea unei singure mașini.

O altă limitare ține de **scalabilitate verticală**: un singur server trebuie dimensionat (CPU, memorie, stocare) pentru a suporta vârful de încărcare al tuturor clienților conectați; acest lucru poate deveni ineficient sau costisitor la scară foarte mare.

Chiar și cu scalare orizontală (mai multe servere), arhitectura rămâne centralizată logic - managementul complex (coordonarea între servere, distribuția sesiunilor, consistența datelor între noduri) revine tot părții de server și adaugă complexitate suplimentară.

În plus, **latența rețelei** este mereu un factor: clientul depinde de calitatea conexiunii la server, ceea ce înseamnă că aplicația trebuie să fie tolerantă la întârzieri sau deconectări (mai ales relevant în cazul clienților mobili cu rețele potențial instabile).

Nu în ultimul rând, securitatea centralizată are și un revers: dacă serverul este compromis (atacat cu succes), atacatorul poate obține acces la toate datele și serviciile, afectând toți clienții. De aceea, se impune o protecție foarte bună a infrastructurii de server.

**Exemple de aplicații client-server:** Arhitectura client-server este folosită pe scară largă în sistemele informatice moderne.

Un exemplu cotidian este **Web-ul** însuși: browser-ul web (clientul) comunică cu serverele web (Apache, Nginx, IIS etc.) pentru a cere pagini sau servicii web – practic fiecare site vizitat implică acest model.

**Sistemele de e-mail** funcționează similar: clientul de e-mail (ex. Outlook, Thunderbird sau aplicația mobilă de mail) se conectează la un server de e-mail (SMTP/IMAP server) pentru a trimite sau prelua mesaje; serverul gestionează căsuțele poștale și distribuie mesajele către destinatarii corespunzători.
De asemenea, **băncile de date centralizate** urmează deseori modelul 2-tier client-server: de pildă, o aplicație desktop de gestiune (client) se conectează direct la un server de baze de date (ex. un server Oracle sau MySQL) și execută interogări SQL.

Multe aplicații de birou mai vechi sau aplicații enterprise simple au fost construite ca *aplicații pe două niveluri* (2-tier), cu interfața și logica la client și cu un server de baze de date în spate. În prezent însă, pe măsura ce aplicațiile au devenit mai complexe și numărul de utilizatori a crescut, s-a trecut tot mai mult la arhitecturi multi-tier, discutate în secțiunea următoare.

## 2.2 Arhitecturi **multi-tier** (pe mai multe niveluri)

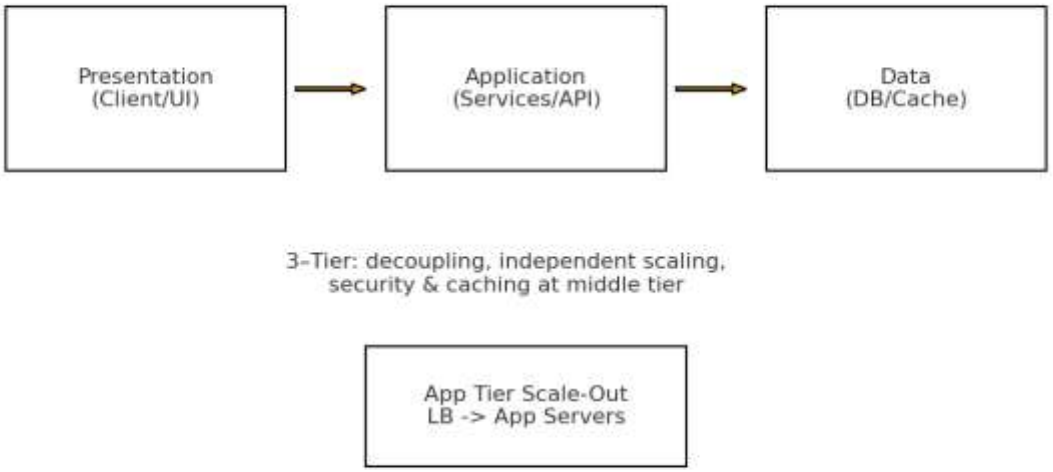


*Figura 2.2: Arhitectura pe 3 niveluri (trei* tiers*): client (prezentare), server de aplicație (logică de procesare) și server de date (SGBD). Fiecare nivel poate fi dezvoltat și scalat separat.*

**Definiție și concepte:** O arhitectură *multi-tier* (cu mai multe niveluri) reprezintă o extindere a modelului client-server prin adăugarea unor **niveluri intermediare** de procesare. Cea mai cunoscută variantă este **arhitectura în 3 niveluri (3-tier)**, care introduce un nivel suplimentar între client și serverul de date. Astfel, sistemul este împărțit în *trei* straturi principale, fiecare cu rol distinct: un strat de **prezentare** (interfața cu utilizatorul, de obicei pe client), un strat de **logică de aplicație** (procesare și reguli de afaceri, pe un server intermediar) și un strat de **date** (stocare și gestionare a datelor, de obicei pe un server de baze de date). Fiecărui strat îi corespunde un anumit tip de componentă software/hardware – de exemplu, într-o aplicație web clasică: browser-ul sau aplicația mobilă reprezintă nivelul de prezentare (**clientul**), un server web sau de aplicații (ex: un server Java EE, un server Node.js sau Python Flask) constituie nivelul de logică (procesând

solicitările, aplicând regulile de afaceri), iar un server de baze de date (ex: Oracle, PostgreSQL, MongoDB etc.) reprezintă nivelul de date, ocupându-se de stocarea informației și execuția interogărilor.

În arhitectura multi-tier, **serverul din modelul client-server este decompus în noduri mai granulare** (mai multe servere specializate)[3].

Această separare are scopul de a decupla responsabilitățile: în loc ca un singur server să facă și logică de afaceri, și acces la BD, și comunicare cu clientul, se *împart sarcinile* între componente diferite.

De exemplu, un **server de aplicații** dedicat poate prelua cererile de la client, poate aplica logica (validări, calcule, fluxuri tranzacționale etc.), apoi comunică cu un **server de baze de date** separat care administrează efectiv datele. Clientul nu mai interacționează direct cu baza de date, ci întotdeauna prin intermediul nivelului de logică (care acționează ca o punte). În plus, se pot adăuga și alte niveluri în funcție de complexitate – de aceea se vorbește în general de arhitecturi *N-tier*, unde *N* poate fi 3 sau mai mare.

Spre exemplu, o arhitectură *four-tier* ar putea separa nivelul de logică de afaceri de nivelul de **acces la date** (data access layer), având astfel: client (UI), server de aplicații (logică), un server intermediar pentru acces la date (de exemplu un API gateway sau microserviciu specializat) și abia apoi serverul de baze de date.

În alte cazuri, peste nivelul de prezentare se mai consideră un nivel de **client extern** – de exemplu, aplicațiile mobile sau frontend-urile web care consumă un *API* REST pot fi văzute ca un nivel distinct față de interfața utilizator tradițională. Indiferent de numărul lor exact, principiul rămâne că fiecare tier (nivel) își asumă un rol bine definit în sistem, comunicând cu cele alăturate prin interfețe/documentații clar specificate.

**Avantaje ale arhitecturilor multi-tier:**

1) Principalul beneficiu al utilizării mai multor niveluri este creșterea **modularității** și a **separării preocupărilor** (*separation of concerns*). Fiecare strat poate fi dezvoltat, mentenanțat și scalat independent, ceea ce aduce mai multă **flexibilitate**. De exemplu, dacă dorim să îmbunătățim performanța bazei de date, putem optimiza sau înlocui doar nivelul de date (ex. schimbând SGBD-ul sau adăugând replici), fără a modifica logica de la nivelul de aplicație sau codul UI de pe client[3]. Similar, dacă dorim să schimbăm interfața cu utilizatorul (să trecem de la o aplicație desktop la una web, sau la una mobilă), putem face acest lucru atâta timp cât noul client folosește aceeași interfață de comunicare cu serverul de aplicații, fără a schimba modul în care funcționează logica de pe server sau schema de date.
2) Un alt avantaj major este **scalabilitatea îmbunătățită**: putem scala orizontal fiecare nivel în mod independent pe baza cererii. Dacă partea de logică devine bottleneck (consumă mult CPU), putem adăuga mai multe servere de aplicație și un load balancer care să distribuie cererile între ele, fără a atinge baza de date sau clienții. Dacă baza de date devine punctul limitativ, putem folosi replicare, sharding sau upgrade hardware la nivelul SGBD, în timp ce serverele de aplicație și clienții rămân neschimbați. Această **scalare independentă pe niveluri** face sistemul multi-tier potrivit pentru aplicații cu foarte mulți utilizatori și volum mare de date[3].
3) De asemenea, **mentenanța** și dezvoltarea în echipă sunt facilitate: echipe diferite se pot ocupa de straturi diferite (ex. o echipă de frontend dezvoltă interfața, o echipă de backend dezvoltă logica de server, iar o echipă de DB admins se ocupă de schema BD

și optimizare). Acest lucru crește productivitatea și permite actualizări mai frecvente – de exemplu, putem lansa o nouă versiune a componentelor de client sau a serverului de aplicații fără a întrerupe întregul serviciu, atâta timp cât interfețele de comunicare (API-urile) rămân compatibile.

Arhitecturile multi-tier aduc și **flexibilitate tehnologică**: fiecare nivel poate folosi un stack tehnologic optim pentru rolul său. De exemplu, front-end-ul poate fi realizat în React/Angular (JavaScript) pentru interfață web sau Kotlin/Swift pentru mobil, serverul de aplicații poate fi scris în Java sau Python, iar baza de date poate fi Oracle sau MongoDB, fără constrângeri reciproce, comunicând prin API-uri sau protocoale bine definite (HTTP/REST, RPC, etc.). În plus, introducerea unui nivel de server de aplicații permite implementarea de mecanisme precum **cache** la nivel de server (ex. memorarea rezultatelor unor interogări frecvente, reducând astfel sarcina pe baza de date) și managementul sesiunilor utilizatorilor într-un mod centralizat (în loc să fie gestionate de fiecare client în parte).

**Provocări și dezavantaje:** Pe lângă beneficii, arhitecturile multi-tier au și dezavantaje de luat în seamă. În primul rând, crește **complexitatea generală** a sistemului – sunt mai multe componente de configurat, dezvoltarea și depanarea devin mai dificile deoarece trebuie să urmărim fluxul de execuție prin mai multe procese/noduri. Comunicarea dintre niveluri introduce și **suprasarcină de rețea și latență**: fiecare apel de la client la serverul de aplicații și apoi de la acesta la baza de date adaugă întârziere comparativ cu un apel local.

De exemplu, o tranzacție complexă poate necesita ca serverul de aplicații să facă mai multe interogări la BD; dacă cele două rulează pe mașini diferite, fiecare interogare implică trafic de rețea. Optimizarea performanței necesită astfel mecanisme precum pooling de conexiuni, reducerea traficului între nivele (ex. prin apeluri *batch* sau proceduri stocate) și cache inteligent.

De asemenea, **gestionarea consistenței datelor** poate fi mai dificilă: într-o arhitectură 2-tier simplă, un singur client poate executa tranzacții direct pe BD; într-o arhitectură multi-tier, dacă introducem cache la nivel de server aplicație sau replici de baze de date pentru scalare, trebuie să asigurăm că toate nivelurile "văd" o versiune coerentă a datelor (de exemplu, invalidarea cache-ului la actualizări, sincronizarea replicilor etc.).

Arhitecturile multi-tier pot suferi de **probleme de compatibilitate** la schimbarea interfețelor: dacă modificăm API-ul dintre client și server sau dintre serverul de aplicații și BD, trebuie să ne asigurăm că toate componentele care interacționează se actualizează corespunzător. Acest lucru poate necesita un control atent al versiunilor și teste riguroase de integrare.

Nu în ultimul rând, *costul* și *resursele* necesare pot crește, deoarece în loc de o singură mașină server, acum pot exista mai multe servere (ex. unul sau mai multe pentru aplicație, unul sau mai multe pentru BD, poate și un server de cache separat etc.). Monitorizarea și orchestrarea acestor componente (logging distribuit, monitorizare metrici, gestionarea failover-ului între multiple instanțe) necesită instrumente dedicate. Practic, obținem un sistem mai robust și scalabil, dar care cere **infrastructură și expertiză suplimentară** comparativ cu o aplicație monolitică 2-tier.

**Exemple și studii de caz moderne:** Aproape orice aplicație web sau mobilă la scară largă din industrie folosește o formă de arhitectură multi-tier. Un exemplu clasic este un site de comerț electronic (**e-commerce**). Să luăm scenariul Amazon.com: inițial, Amazon a pornit cu o arhitectură pe 2 niveluri (o aplicație monolit care rula pe un server și folosea o bază de date) – pe măsură ce numărul de utilizatori și volumul de date au explodat, această abordare nu a mai făcut

față. Amazon a migrat în anii 2000 spre o arhitectură de tip *service-oriented* (SOA) și ulterior microservicii, care poate fi privită ca o extindere extremă a conceptului multi-tier. Astăzi, website-ul Amazon este deservit de **sute de servicii backend specializate**, fiecare responsabil de un sub-domeniu (serviciu de plăți, serviciu de recomandări, serviciu de gestionare stocuri etc.), orchestrate împreună[4]. Când un utilizator încarcă pagina unui produs pe Amazon, în spate se fac *100-150 de apeluri interne* către diverse servicii pentru a compune informațiile (preț, stoc, recenzii, recomandări de produse similare ș.a.)[4]. Aceste servicii stochează date în baze de date diferite și comunică prin API-uri; practic, arhitectura Amazon este multi-tier distribuită pe steroizi – cu un nivel de prezentare (website-ul și aplicația mobilă), un nivel intermediar compus din numeroase servicii de aplicație, și multiple niveluri de date (baze de date și sisteme de cache). **Netflix** este un alt exemplu celebru: pentru a servi streaming video către milioane de utilizatori, Netflix a adoptat o arhitectură microservicii (care poate fi văzută ca N-tier) unde fiecare funcționalitate (catalog de filme, recomandări, autentificare utilizator, transcodare video etc.) este un serviciu separat distribuit global[5]. Ei au peste o mie de microservicii care cooperează pentru a oferi experiența finală, și folosesc rețele de distribuție de conținut (CDN) ca un strat suplimentar pentru livrarea rapidă a conținutului video către client (un alt tier specializat).

Chiar și în contexte enterprise mai tradiționale, **aplicațiile pe 3 niveluri** rămân un standard. De exemplu, o aplicație internă de management al resurselor (ERP) poate avea: un client web sau desktop (prezentare), un server de aplicație unde rulează regulile de business (posibil un server Java EE/.NET care expune servicii web pentru client) și un server de baze de date (Oracle/SQL Server) unde se stochează datele companiei.

**Platformele web moderne** (de la rețele sociale la aplicații bancare online) folosesc și ele arhitectura multi-tier: de obicei un front-end (browser sau app mobil) comunică cu un *backend* format din API-uri web (REST/GraphQL) care la rândul lor accesează baze de date și alte servicii. Acest model permite, de exemplu, ca aceeași logică de business să fie accesată atât de site-ul web, cât și de aplicația mobilă și eventual de clienți terți (prin expunerea unui API public), fără duplicarea codului.

Nu în ultimul rând, conceptul de **microservicii** menționat – deși este o arhitectură distinctă – poate fi privit ca o continuare a filosofiei multi-tier: se împarte monolitul aplicației în *și mai multe* componente decuplate. Fiecare microserviciu are propriul *tier* de logică și de date, comunicând cu celelalte prin interfețe bine definite. Avantajul este o granularitate și mai mare (fiecare serviciu poate fi dezvoltat/scalat independent), însă prețul este creșterea complexității infrastructurii și a cerințelor de orchestrare (sisteme de *containerizare*, servicii de descoperire, gateway-uri API, etc.).

În concluzie, arhitecturile multi-tier (în special cele 3-tier) reprezintă backbone-ul aplicațiilor distribuite moderne, oferind un compromis eficient între performanță, scalabilitate și ușurința de dezvoltare în echipă.

2.3 Arhitecturi **peer-to-peer** (P2P)

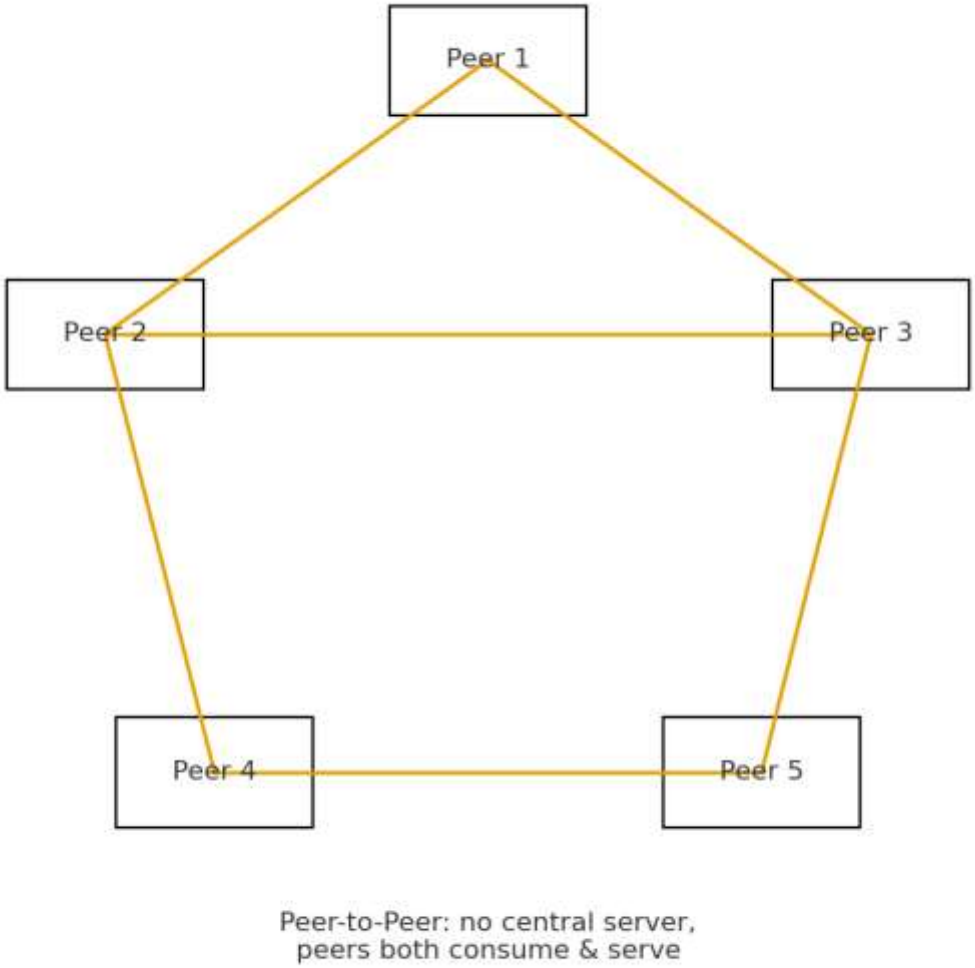


*Figura 2.3: Ilustrarea unei rețele* peer-to-peer *nestructurate. Nodurile (calculatoare personale, laptop-uri sau dispozitive mobile) comunică între ele direct, în mod ad-hoc, fără a exista un server central. Conexiunile se stabilesc în funcție de disponibilitatea nodurilor, formând o topologie dinamică.*

**Prezentare și caracteristici:** Arhitectura *peer-to-peer (P2P)* este un model distribuit în care nu există roluri fixe de client și server – *fiecare nod din rețea este simultan și client, și server*, putând atât să ceară, cât și să furnizeze servicii sau resurse. Cu alte cuvinte, în locul unei structuri centralizate, P2P propune o **rețea de noduri egale (peers)**, care comunică direct unele cu altele. Fiecare peer deține o parte din resursele sistemului și poate iniția sau răspunde la cereri de la alte peer-uri. Acest model elimină complet dependența de un nod central coordonator, distribuind responsabilitățile între participanți în mod democratic.

Principalele caracteristici ale arhitecturilor peer-to-peer sunt legate de **decentralizare** și **auto-organizare**. Pentru început, *nu există un punct unic de control*: rețeaua se menține prin cooperarea nodurilor, care pot intra sau ieși oricând.

Aceasta conferă un grad mare de **robustețe și toleranță la defecțiuni** – sistemul P2P continuă să funcționeze chiar dacă unele noduri cad, deoarece altele pot prelua sarcinile (atâta timp cât există suficiente legături între nodurile rămase).

De asemenea, arhitecturile P2P sunt în general **scalabile natural**: pe măsură ce noi noduri se alătură rețelei, ele aduc cu sine resurse suplimentare (putere de procesare, stocare, lățime de bandă), astfel încât capacitatea totală a sistemului crește odată cu numărul de participanți. Acest lucru contrastează cu modelul client-server, unde adăugarea de clienți crește sarcina pe serverul central – în P2P, noile noduri preiau și ele din sarcină.

Un alt avantaj este **sharing-ul eficient al resurselor**: fiecare peer poate oferi ceea ce are – fie că vorbim de fișiere partajate, putere de calcul neutilizată sau date – contribuind la ansamblu. În anumite aplicații, P2P poate reduce și costurile, deoarece nu necesită infrastructură centrală dedicată (fiecare participant folosește propriul hardware și conexiune).

În arhitectura P2P, un nod tipic conține toate straturile aplicației (prezentare + logică + date) în cadrul lui[6]. Cu alte cuvinte, **fiecare peer este o instanță completă a aplicației** care își gestionează atât interfața (dacă există una, de ex. un UI), cât și procesarea datelor locale și stocarea parțială a datelor sistemului. Peers comunică între ele pentru a sincroniza date sau pentru a solicita informație pe care nu o au local. De multe ori, fiecare nod menține o **copie (integrală sau parțială) a stării globale** a sistemului sau poate accesa această stare prin intermediul altor noduri[6]. De exemplu, într-o rețea de file sharing P2P, fiecare computer stochează anumite fișiere și are o listă a fișierelor disponibile în rețea (sau a nodurilor cunoscute care dețin fragmente). Dacă un nod are nevoie de un fișier pe care nu îl are local, va căuta în rețea alt nod care îl are și îl va descărca direct de la acela, în loc să meargă la un server central.

**Tipuri de rețele P2P:** Există două mari categorii de rețele peer-to-peer: *nestructurate* și *structurate*.

În rețelele **P2P nestrucrurate**, conexiunile între noduri sunt formate ad-hoc, aleator sau bazat pe criterii locale (ex: cunoștințele unui nod, ultimele întâlniri etc.). Aceste rețele nu au un algoritm global de plasare a datelor, ceea ce înseamnă că găsirea unui anumit element poate implica *căutare prin inundare* (flooding) – un nod întreabă vecinii, care la rândul lor întreabă pe ai lor, și așa mai departe, până se găsește resursa. Avantajul e simplitatea și robustețea la noduri dinamice, dar dezavantajul e că operațiile de căutare pot fi ineficiente (generând mult trafic).

În rețelele **P2P structurate**, se folosește o funcție de distribuție a conținutului: de obicei un *Distributed Hash Table (DHT)* care map-ează fiecare resursă la un nod responsabil pe baza unui hash. Astfel, fiecare nod știe relativ *unde* ar trebui să fie o anumită informație și poate ruta cererea eficient prin rețea (logaritmic cu numărul de noduri, de ex.). Rețele structurate precum Chord, Kademlia, Pastry au algoritmi care asigură localizarea rapidă a datelor, însă sunt mai puțin flexibile la schimbări foarte frecvente ale nodurilor (totuși includ mecanisme de update ale DHT-ului când noduri intră sau ies).

Multe sisteme P2P moderne combină elemente din ambele - de exemplu protocolul BitTorrent folosește trackere (centralizate inițial sau DHT distribuit) pentru a ajuta la găsirea peer-urilor care au un anumit fișier, după care transferul efectiv are loc în mod P2P necentralizat.

**Avantaje ale P2P:** Așa cum am menționat, lipsa unui server central conferă P2P **toleranță ridicată la căderi** – nu există un singur punct a cărui defecțiune să oprească întregul sistem. Dacă

un nod din Figura 2.3 de mai sus părăsește rețeaua, altele pot comunica pe rute alternative. De asemenea, arhitectura P2P excelează la **scalabilitate**: în teorie, cu cât ai mai multe noduri, cu atât ai mai multă capacitate totală. Acest lucru s-a văzut practic în sisteme de partajare fișiere – rețelele P2P au susținut milioane de utilizatori simultan fără a necesita centre de date costisitoare. Un alt avantaj este **eficiența distribuției**: de exemplu, în distribuția de fișiere mari (video, ISO etc.), modelul P2P (ca BitTorrent) permite ca utilizatorii să se descarce unii de la alții în paralel, ceea ce scalează mult mai bine decât să descarce toți de la un server unic (unde lățimea de bandă ar fi un gât de sticlă). În plus, P2P poate oferi un grad sporit de **anonimitate sau descentralizare** dorit în unele aplicații: de exemplu rețelele de criptomonede (blockchain) se bazează pe P2P tocmai pentru a evita controlul centralizat – niciun server sau entitate unică nu deține "puterea", ci toate nodurile împreună mențin consensul.

**Dezavantaje și probleme în P2P:** În ciuda punctelor forte, P2P are și puncte slabe semnificative.

**Lipsa de control central** face dificilă administrarea rețelei și aplicarea politicilor globale. Sarcini precum asigurarea securității, integrității datelor sau aplicarea unor reguli (de ex. filtrarea conținutului necorespunzător) devin complicate când fiecare nod poate acționa independent.

Una dintre marile probleme în P2P este **managementul încrederii și securității**: cum știi că un peer este de încredere? În rețele centralizate, serverul poate autentifica utilizatorii, poate verifica permisiuni; în P2P, nu există implicit o autoritate de autentificare, deci se recurge la certificate distribuite, sisteme de reputație sau protocoale criptografice complexe pentru a preveni acțiuni malițioase (cum ar fi un nod care furnizează date corupte sau interferează cu comunicarea).

Un alt aspect dificil este **consistența datelor**: dacă fiecare peer deține o copie a unor părți din date, cum ne asigurăm că toate copiile ajung eventual să fie sincronizate? Protocolul de gossip (bârfă) este o abordare des folosită, unde nodurile își comunică periodic starea unii altora pentru a converge către o stare comună, însă acest lucru poate cauza latențe în propagarea datelor (consistență eventuală). În aplicații critice (ex. tranzacții financiare), astfel de întârzieri pot fi problematice.

P2P suferă uneori și la capitolul **performanță individuală**: deoarece nu există un nod super-puternic care să proceseze cererile, viteza și calitatea serviciului depind de nodurile voluntare din rețea, care pot avea capacități variabile. De exemplu, într-o rețea P2P de streaming video, calitatea fluxului poate depinde de ce vecini ai și cât de bună e conexiunea lor.

Mai mult, topologia P2P poate duce la **supraîncărcarea unor noduri**: dacă anumite noduri devin populare (au un conținut foarte cerut), ele pot fi solicitate intens de altele, consumându-le toată lățimea de bandă sau puterea de calcul – fără un load balancer central, acest lucru poate duce la dezechilibre (unele noduri suprasolicitate, altele nefolosite). Mecanismele de *choking/unchoking* din BitTorrent sau algoritmii de balansare a sarcinii în DHT-uri încearcă să abordeze această problemă, dar nu o elimină complet.

**Exemple de aplicații peer-to-peer:**

Multe dintre tehnologiile populare de partajare de informație și colaborare descentralizată sunt bazate pe P2P. Un exemplu clasic este **sistemul BitTorrent** de partajare a fișierelor: când un utilizator dorește să descarce un fișier mare, torrent-ul îi permite să ia bucăți de la mai mulți alți utilizatori (peers) care deja au descărcat (sau sunt la rândul lor în proces). Fiecare peer care are bucăți devine un server pentru alții (seeder), în timp ce descarcă alte bucăți ca client (leecher). Acest model distribuie sarcina foarte eficient, astfel încât chiar și fișiere de mari dimensiuni pot fi propagate la scară mare fără un server central de fișiere. Protocolul utilizează inițial trackere sau

DHT pentru a conecta nodurile care au același torrent (identificat printr-un hash), apoi comunicarea efectivă este P2P (protocol BitTorrent peste TCP/UDP).

O altă categorie importantă sunt **rețelele de criptomonede și blockchain** (Bitcoin, Ethereum etc.), care sunt construite ca rețele P2P. Aici fiecare nod din rețea menține o copie a registrului (blockchain-ului) și validează tranzacții și blocuri. Nodurile comunică între ele pentru a transmite noile tranzacții și pentru a ajunge la consens asupra următorului bloc adăugat, folosind algoritmi precum Proof-of-Work sau Proof-of-Stake. Decentralizarea asigură că *nicio entitate unică nu controlează moneda*, iar rețeaua continuă să funcționeze atâta timp cât există noduri oneste care să proceseze tranzacții. Reziliența este foarte mare – de exemplu, rețeaua Bitcoin (cu zeci de mii de noduri global) ar necesita compromiterea majorității puterii de calcul pentru a fi oprită sau fraudată, lucru considerat extrem de improbabil cu tehnologia actuală.

**Comunicare și colaborare P2P:**

Aplicațiile de tip **Voce/IP și mesagerie** au folosit și ele arhitecturi P2P în trecut. Cel mai cunoscut caz este **Skype** (versiunile anterioare achiziției de către Microsoft): Skype folosea o rețea P2P pentru a conecta utilizatorii în apeluri audio/video. Folosea noduri *super-peers* (unii utilizatori cu conexiuni bune erau promovați ca noduri intermediare) pentru a ruta apelurile prin Internet fără să depindă de un server central pentru datele din apel (serverele centrale erau folosite doar pentru autentificare și publicitate a prezenței). Astfel, calitatea apelului și conectivitatea erau menținute chiar dacă segmente ale rețelei întâmpinau probleme, iar costurile de infrastructură erau reduse (traficul de voce trecea prin conexiunile utilizatorilor). După 2011, Skype și alte servicii similare au migrat totuși parțial spre arhitecturi hibride, cu mai mult control central, din motive de securitate și conformitate.

Un alt exemplu modern este **aplicația de partajare de fișiere distribuită IPFS (InterPlanetary File System)** – un protocol și rețea P2P pentru stocarea și partajarea de date într-un sistem de fișiere distribuit. IPFS permite adresarea conținutului printr-un hash și obținerea lui de la oricare nod care îl deține, fără nevoie de un server central de hosting. Este folosit ca infrastructură pentru web distribuit (de exemplu, unele rețele de socializare descentralizate sau aplicații de tip Web3 stochează conținutul media în IPFS pentru reziliență și evitare cenzură).

În domeniul **bazelor de date distribuite**, există sisteme care adoptă arhitectura P2P la nivelul nodurilor de stocare. Un exemplu este **Apache Cassandra**, o bază de date NoSQL de tip wide-column. Cassandra folosește un model total distribuit: toate nodurile din cluster sunt egale (nu există un master central), comunică printr-un protocol de *gossip* și folosesc o variantă de DHT (bazat pe token ring) pentru distribuirea datelor. Astfel, Cassandra obține scalare liniară adăugând noduri, iar toleranța la defecțiuni este ridicată – orice nod poate răspunde la cereri pentru că datele sunt replicate și rețeaua știe cum să găsească partițiile necesare. Acest design P2P elimină un singur punct de eșec și este potrivit pentru aplicații care necesită *high availability* pe mai multe centre de date. Desigur, implementarea este complexă: asigurarea consistenței între replici se face prin protocolul **gossip/antientropy** și **consistență eventuală** (cu posibilitatea de a configura niveluri de consistență la citire/scriere).

În concluzie, arhitectura peer-to-peer oferă o alternativă puternică la modelele client-server centralizate, în special acolo unde se dorește distribuirea sarcinilor și eliminarea dependențelor de un nod central. P2P stă la baza multor inovații moderne (de la sharing economy în IT, la tehnologii blockchain) și extinde orizontul aplicatiilor distribuite. Pe de altă parte, dezvoltarea de aplicații P2P necesită abordarea unor probleme dificile de consens, securitate și eficiență a căutării, motiv pentru care adesea se recurge la **arhitecturi hibride**: de exemplu, multe aplicații “de masă”

folosesc un server central pentru autentificare sau indexare, dar transferul de date efectiv îl fac P2P (un compromis între control și scalare). Alegerea arhitecturii potrivite (client-server, multi-tier sau peer-to-peer) depinde de cerințele sistemului - adesea, componente diferite ale aceluiași sistem pot folosi modele diferite (ex: un joc online poate folosi client-server pentru login și scoruri, dar P2P pentru comunicarea dintre jucători în sesiune). Important este ca un arhitect de sistem să cunoască avantajele și dezavantajele fiecărui model, astfel încât să poată combina optim tehnologiile pentru cerințele aplicației respective.

**Bibliografie:**

[1] Client–server model - Wikipedia
https://en.wikipedia.org/wiki/Client%E2%80%93server_model
[2] [3] [6] [8] What is a distributed system? | Atlassian
https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture
[4] [5] Microservice Architecture pattern
https://microservices.io/patterns/microservices.html
[7] Distributed Systems 3rd edition (2017) - DISTRIBUTED-SYSTEMS.NET
https://www.distributed-systems.net/index.php/books/ds3/