7 Concepte generale privind sistemele distribuite

Vom începe acest capitol prin a introduce succint principale concepte şi tendinţe cu privire la sistemele distribuite. În literatura de specialitate s-au încercat, de-a lungul anilor, diverse definiţii privind sistemele distribuite, însă nici una nu s-a dovedit a fi întru totul satisfăcătoare, suficient de acoperitoare şi fără a fi în contradicţie cu vreuna dintre celelalte.

Pentru nevoile acestei prezentări vom utiliza definiţia dată de Tanenbaum în monografia sa asupra sistemelor distribuite[1]. Conform acestei definiţii, un sistem distribuit este o colecţie de computere independente care reuşesc să fie percepute de către utilizatorii umani ca un sistem unic şi coerent. Această definiţie are două aspecte. Primul aspect priveşte partea de hardware şi se referă faptul că maşinile de calcul sunt văzute ca multitudine de entităţi autonome. Cel de al doilea aspect priveşte partea de software şi se referă la faptul că utilizatorii trebuie să perceapă ansamblul de programe ca pe un singur sistem. Ambele aspecte sunt deopotrivă importante şi trebuie să fie realizate împreună.

Se pot distinge câteva caracteristici comune tuturor sistemelor distribuite şi acestea sunt de natură să suplinească o definiţie mai complicată. Una dintre caracteristicile importante ale sistemelor distribuite este aceea că diferenţa dintre diversele platforme de calcul şi maniera de comunicare între acestea este transparentă pentru utilizator. Aceeaşi caracteristică o posedă şi organizarea internă, la nivel logic, a unui sistem distribuit.

O altă caracteristică este aceea că utilizatorii şi alte aplicaţii din afara sistemului pot interacţiona cu un sistem distribuit într-o manieră uniformă şi coerentă, indiferent unde şi când are loc această interacţiune.

De asemenea, sistemele distribuite trebuie să fie uşor de extins şi de redimensionat. Această caracteristică este o consecinţă directă a faptului că este vorba de o colecţie de maşini de calcul dar, în acelaşi timp, şi o consecinţă a faptului că această multitudine de computere care formează sistemul, interacţionează între ele într-un mod transparent pentru utilizator, creând impresia de întreg.

O altă caracteristică este aceea ca un sistem distribuit va fi întotdeauna disponibil spre a fi exploatat de utilizatori, chiar dacă anumite componente ale acestuia pot să nu fie în funcţiune la anumite momente de timp. Utilizatorii şi celelalte aplicaţii care interacţionează cu un sistem distribuit nu trebuie să sesizeze că anumite componente au fost debranşate, înlocuite cu altele, sau noi componente au fost adăugate în sistem pentru a satisface noi cerinţe sau pentru a deservi mai mulţi utilizatori sau mai multe aplicaţii.

Pentru a putea satisface aceste cerinţe şi pentru asigurarea interconectării unui hardware de multe ori eterogen, sistemele distribuite sunt adesea organizate prin intermediul unui nivel software plasat, la nivel logic, între componentele de nivel înalt, care asigură interfaţa cu utilizatorii şi nivelul de jos, constituit de sistemul de operare, ***Figura 7.1***. De aceea acest nivel, care joacă rolul unei platforme de comunicaţie pentru sistemul distribuit, poartă denumirea de *middleware*.

[1] În capitolul introductiv din **[TANEN-DIS]**

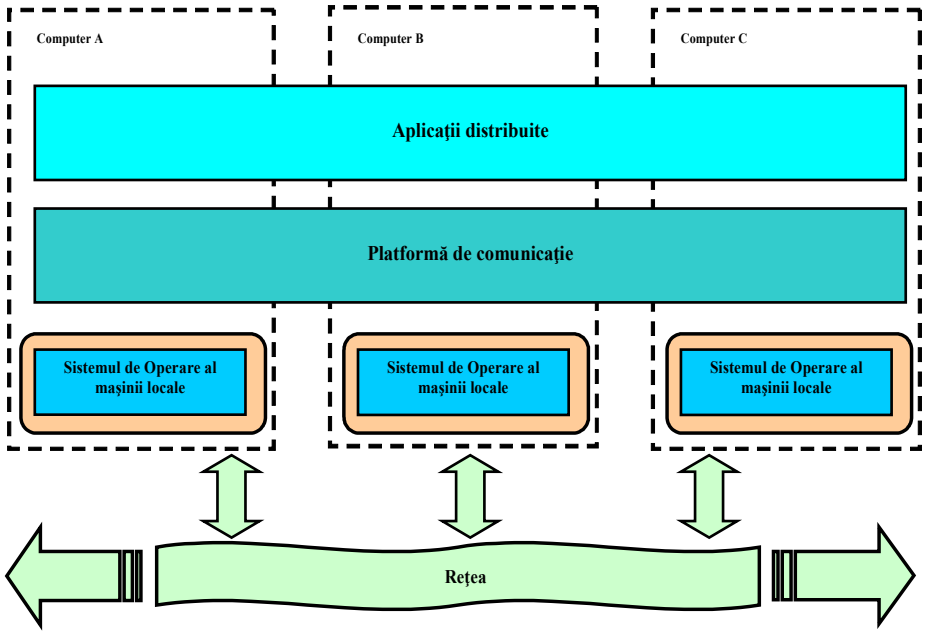

Figura 7.1 – Arhitectura unui sistem distribuit cu platformă unică de comunicație

Un sistem distribuit este menit să conecteze în mod facil resursele de calcul de utilizatorii acestora. De asemenea, trebuie să mențină transparent pentru utilizator faptul că aceste resurse sunt distribuite fizic într-o rețea. Trebuie să fie un sistem deschis, care să accepte extinderi ulterioare și să fie poată fi cu ușurință redimensionat.
În ceea ce privește transparența, acest concept se poate aplica mai multor aspecte ale sistemelor distribuite:

- accesul – transparență în ceea ce privește diferența dintre modul de reprezentare și organizare a datelor, precum și maniera în care acestea, ca și resurse, sunt accesate;
- localizarea – transparență în ceea ce privește distribuția fizică a componentelor sistemului;
- migrarea – transparența faptului că o resursă poate fi mutată fizic dintr-un loc într-altul;
- re-localizarea – transparența mutării unei resurse dintr-un loc într-altul, în timp ce resursa este utilizată în sistem;
- replicarea – transparența procesului de replicare a unei resurse în cadrul sistemului;
- concurența – transparență în ceea ce privește situația în care o resursă este utilizată (partajată) în mod concurențial de mai mulți utilizatori;
- indisponibilitatea unei resurse – este transparent faptul că o anumită resursă din sistem nu mai poată fi accesată și că se urmează o procedură de recuperare a datelor;
- persistența – transparență în ceea ce privește stocarea unei resurse în memorie sau pe disc.

În ceea ce privește posibilitățile de redimensionare a unui sistem distribuit, existența a diferite forme de centralizare în sistem poate ridica probleme extrem de complexe când se pune problema redimensionării acestuia. Centralizarea se poate manifesta la nivel de:

- servicii – un singur server pentru mai mulți utilizatori;
- date – o unică bază de date în sistem;
- algoritmi – realizarea de rutări pe baza unei informații complete despre sistem.

Fiecare dintre aceste situații își poate pune amprenta într-o mai mică sau mai mare măsură asupra ușurinței cu care un sistem distribuit poate fi ajustat în dimensiuni, însă dintre toate, cele mai complicate și în consecință, cele mai dificil de surmontat sunt situațiile în care se proiectează sistemul utilizându-se algoritmi centralizați. De aceea apare ca o necesitatea, încă din faza de proiectare, utilizarea de algoritmi descentralizați pentru realizarea funcțiilor sistemului. Acești algoritmi au următoarele caracteristici, care îi disting de algoritmii centralizați:

- nici un computer din sistem nu deține o informație completă relativă la starea sistemului;
- fiecare mașină din sistem ia decizii bazându-se numai pe informația locală pe care o deține;
- căderea unei mașini din sistem nu conduce la eșecul algoritmului;
- nu se face nici o asumare implicită relativă la faptul că ar exista un timp global și în consecință, unic pentru întreg sistemul.

7.1 Concepte cu privire la hardware distribuit

Chiar dacă toate sistemele distribuite presupun utilizarea mai multor unități de procesare (CPU), există numeroase modalități de organizare a acestor unități, în special în ceea ce privește modul în care ele sunt interconectate și maniera în care comunică între ele.
În continuare, vom trece succint în revistă, modul cum unitățile de procesare pot fi interconectate. Putem face o primă distincție între sistemele de calcul, după modul în care memoria este utilizată de procesoare:

- computere cu memorie partajată, numite și multiprocesoare;
- computere care nu partajează memorie, numite și multicomputere.

Diferența esențială este că în cazul multiprocesoarelor există un unic spațiu de adrese de memorie care este partajat de către toate unitățile de prelucrare (CPU), în timp ce într-un multicomputer fiecare CPU are propria lui memorie privată.
Sistemele multiprocesor au în comun această caracteristică cheie: toate unitățile de procesare au acces direct la memoria partajată a sistemului. Din moment ce este vorba de o memorie unică, dacă un CPU A scrie un cuvânt în această memorie comună și, o fracțiune se secundă mai târziu, un CPU B citește acest cuvânt din memorie, atunci valoarea cuvântului este cea care tocmai a fost modificată de procesorul A. O unitate de memorie care deține această proprietate se spune că este coerentă. Dacă însă unitățile de procesare sunt conectate printr-o magistrală comună, pe care se află, de asemenea, și unitatea de memorie, atunci această magistrală poate ajunge să fie supraîncărcată și performanța întregului sistem poate scădea în mod drastic. De aceea, ca soluție la această problemă, fiecărei unități de procesare i se adaugă o memorie *cache* de mare viteză. Această

memorie, disponibilă fiecărui procesor, păstrează cuvintele care au fost cel mai recent accesate de către acesta. Toate cererile procesorului, adresate memoriei partajate, trec prin acest *cache*. Dacă cererea își găsește răspunsul în memoria *cache*, atunci magistrala este scutită de traficul către și dinspre memoria comună, partajată. Dacă memoria *cache* este suficient de mare, atunci probabilitatea de succes a regăsirii datelor în *cache* este ridicată și traficul pe magistrală va fi considerabil redus, permițând o configurație cu mai multe procesoare în sistem.
Cu toate acestea, introducerea memoriei *cache* pentru fiecare procesor creează, de asemenea, o altă problemă. Presupunând că două unități de procesare, A și B, citesc amândouă o aceeași locație de memorie, atunci valoarea acesteia în memoria *cache* proprie a fiecărui procesor. Dacă ulterior CPU A suprascrie această locație de memorie, când CPU B citește din nou locația de memorie în cauză, va citi veche valoarea din propriul *cache*, și nu valoarea ce tocmai a fost actualizată de CPU A. Memoria devine în acest fel incoerentă, iar sistemul foarte dificil de programat. De aceea alte măsuri vor fi necesare pentru asigurarea coerenței memoriei sistemului și au fost imaginate alte configurații pentru interconectarea unităților de procesare și a memoriei comune, utilizându-se comutatoare în hardware pentru ordonarea accesului la memoria partajată[1].

Față de multiprocesoare, multicomputerele sunt relativ ușor de construit. Acestea sunt în mod esențial o colecție de mașini, în care fiecare CPU are acces direct la propria sa memorie locală. Principala problemă este aceea a comunicației între aceste computere independente. Din moment ce traficul va fi între computere (*CPU–to-CPU*), volumul vehiculat va fi semnificativ mai redus decât în cazul multiprocesoarelor, cu un trafic intens între unitatea de prelucrare și memorie (*CPU-to-memory*). Vom face din nou distincția între sisteme interconectate printr-o magistrală comună și sisteme bazate pe comutatoare (*switches*). În cazul unui multicomputer bazat pe o magistrală, procesoarele sunt conectate într-o rețea, cum ar cea Ethernet. Lățimea de bandă a unei astfel de rețele este de obicei de 100 Mbps (sunt tot mai răspândite rețelele cu lățimi de bandă de 1Gbps și 10Gbps). În mod similar cu magistrala de memorie a multiprocesoarelor, multicomputerele bazate pe o rețea configurată ca o magistrală devin din ce în ce mai puțin performante, concomitent cu creșterea numărului de computere din rețea. Performanța poate scădea dramatic în cazul sistemelor cu mai mult de 25-100 de noduri.

În cazul unui multicomputer bazat pe comutatoare (*switch-based*), mesajele vehiculate între procesoare sunt rutate prin intermediul rețelei de interconectare și nu publicate pentru toate nodurile (broadcast), ca în cazul sistemelor bazate pe magistrală. Am fost propuse și construite multiple topologii, cele mai populare fiind cele cu topologie grilă și hipercub (***Figura 7.2***). Un hipercub este un cub cu ***n*** dimensiuni. După cum se poate observa din figură, extinderea unui hipercub este relativ ușoară. În exemplul nostru, trecerea de la un hipercub cu 4 dimensiuni la unul cu 5 dimensiuni, presupune adăugarea unei noi perechi de cuburi interconectate cu cea existentă, realizându-se câte încă două conexiuni pentru fiecare nod.

Multicomputerele bazate pe multicast se prezintă într-un spectru foarte larg de implementări. La un capăt al spectrului se întâlnesc supercomputere conținând mii de CPU interconectate prin intermediul unei rețele dedicate, de foarte mare viteză (lățime mare de bandă și latență scăzută) – *Massively Parallel Processors* (MPP). În această abordare, măsuri speciale trebuie luate în ceea ce privește toleranța la erori și căderi ale unor

[1] Mai multe detalii în **[TANEN-DIS]** – Harware Concepts

procesoare, astfel încât întreaga mașină să continue să funcționeze chiar dacă sunt procesoare indisponibile în sistem.

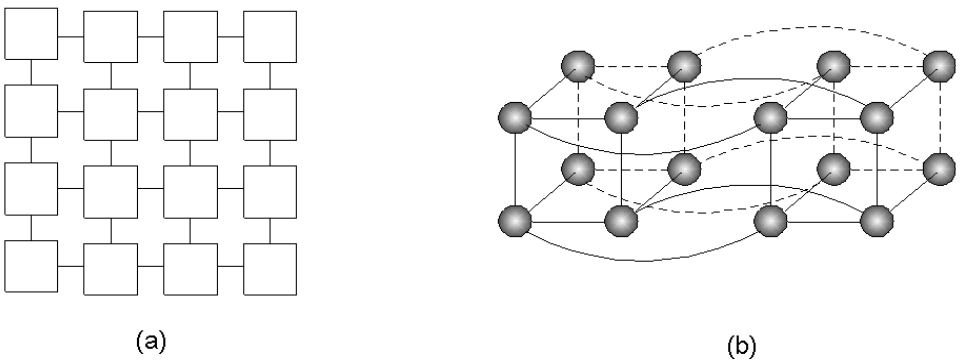

Figura 7.2 – Multicomputer în topologie grilă (a) și în topologie hipercub cu 4 dimensiuni (b)

La celălalt capăt al spectrului se găsește o abordare foarte populară, în care se utilizează practic computere personale obișnuite, conectate între ele prin intermediul unor plăci de rețea ca cele Myrinet, având un software specializat pentru realizarea schimbului de pachete *point-to- point* în această rețea – *Clusters of Workstations* (COW).
Configurațiile de multicomputere prezentate mai sus au nodurile compuse din computere cu performanțe similare sau relativ apropiate, în ceea ce privește tipul de procesor, dimensiunea memoriei, lățimea de bandă pentru operațiile de I/O. În practică însă, multe sisteme distribuite se construiesc într-un mediu multicomputer eterogen. În această situație, măsuri suplimentare de asigurarea a coerenței sistemului trebuie luate și implementate la nivelul programelor care compun sistemul distribuit.

7.2 Concepte cu privire la software distribuit

Așa cum am văzut anterior, componentele fizice sunt importante pentru sistemele distribuite, însă programele care rulează în aceste sisteme sunt cele care determină caracteristicile intrinseci ale acestora. Software-ul unui sistem distribuit are rolul de manager al resurselor fizice și este componenta care asigură transparența interconectării și interacțiunii dintre computere, oferind imaginea unei mașini virtuale care execută anumite sarcini ca o entitate unică.
Sistemele de operare pentru computere distribuite pot fi clasificate principial în două mari categorii[1]:

- sisteme de operare cu interacțiuni foarte strânse între componente (*tightly-coupled*);
- sisteme de operare în care interacțiunile între componente sunt mai relaxate (*loosely-coupled*).

În primul caz, sistemul de operare încearcă, în mod esențial, să mențină o viziune unică, globală a resurselor pe care le administrează. În cel de al doilea caz, este mai curând vorba de o colecție de computere interconectate, fiecare având propriul său sistem de

[1] Conform clasificării făcute de Tanenbaum în **[TANEN-DIS]** – Software Concepts

operare. Cu toate acestea, sistemele de operare ale computerelor dintr-un astfel de sistem deschis, sunt interconectate și lucrează împreună pentru furnizarea unui anumit serviciu, punându-și fiecare resursele proprii la dispoziția celorlalte computere din sistem.
Dacă punem în corespondență această clasificare cu topologiile hardware trecute în revistă anterior, atunci vor vedea că un sistem de operare puternic închegat înseamnă un sistem de operare distribuit (*Distributed Operating Suystem* – DOS), care este utilizat pentru administrarea unui sistem multiprocesor sau multicomputer omogen. Prin contrast, un sistem de operare care nu este destinat interconectării foarte strânse a resurselor care compun sistemul este un sistem potrivit pentru administrarea de colecții de computere eterogene (*Network Operating System* – NOS).
În fapt, progresele făcute în domeniul serviciilor oferite de sistemele de administrare a rețelelor de calculatoare furnizează tocmai această dorită transparență în ceea ce privește distribuția fizică a resurselor. Aceste progrese conduc la apariția unui nou nivel logic de asigurare a interconectării între sistemele de calcul distribuite și neomogene, componentă numită *middleware*.

La rândul lor, sistemele de operare distribuite (DOS) pot fi împărțite în sisteme de operare multiprocesor, care administrează resursele unui sistem fizic multiprocesor și sisteme de operare multicomputer, care sunt destinate administrării unei colecții omogene de computere (***Figura 7.3***).

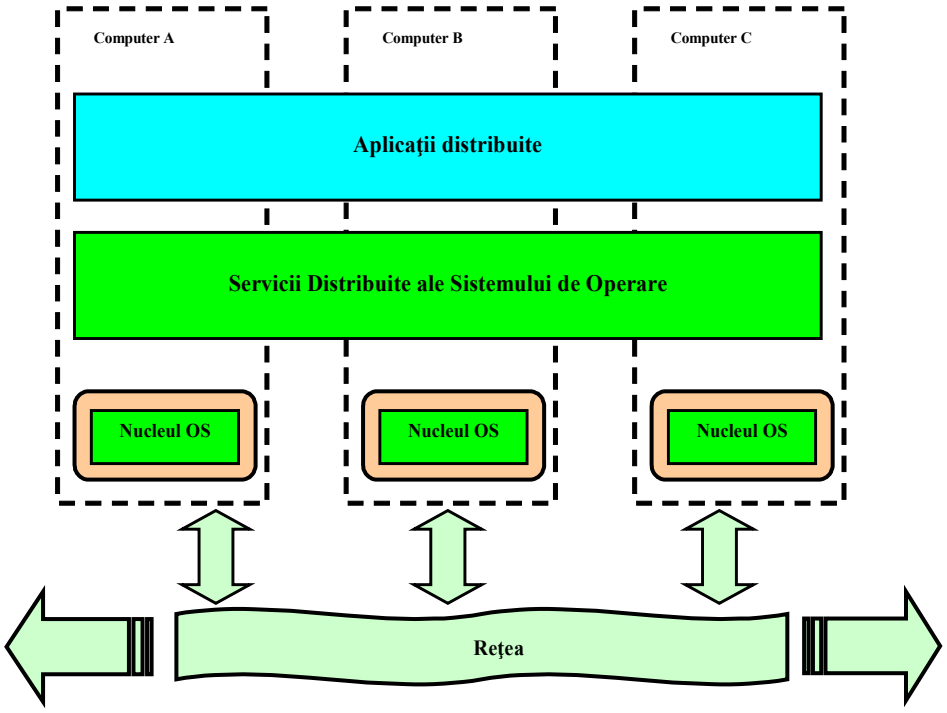

Figura 7.3 – Structura de ansamblu a unui sistem de operare multicomputer

Sistemele de operare pentru multicomputere au o structură și o complexitate cu totul diferite, comparativ cu sistemele de operare pentru multiprocesoare. Diferența este dată de faptul că structurile de date destinate administrării resurselor sistemului, văzut ca un întreg, nu mai pot fi plasate într-o memorie comună partajată de procesoare. În schimb, singurul mijloc de comunicare apare ca fiind transferul de mesaje între computere.

Prin contrast, sistemele de operare în rețea nu asumă că nivelul fizic al mașinilor de calcul ar fi omogen și prin urmare administrabil ca și un singur sistem. Dimpotrivă, premisa este că aceste sisteme sunt construite dintr-o colecție de mașini uniprocesor (pot fi la fel de bine și multiprocesor), fiecare având un sistem propriu de operare, ***Figura 7.4***.

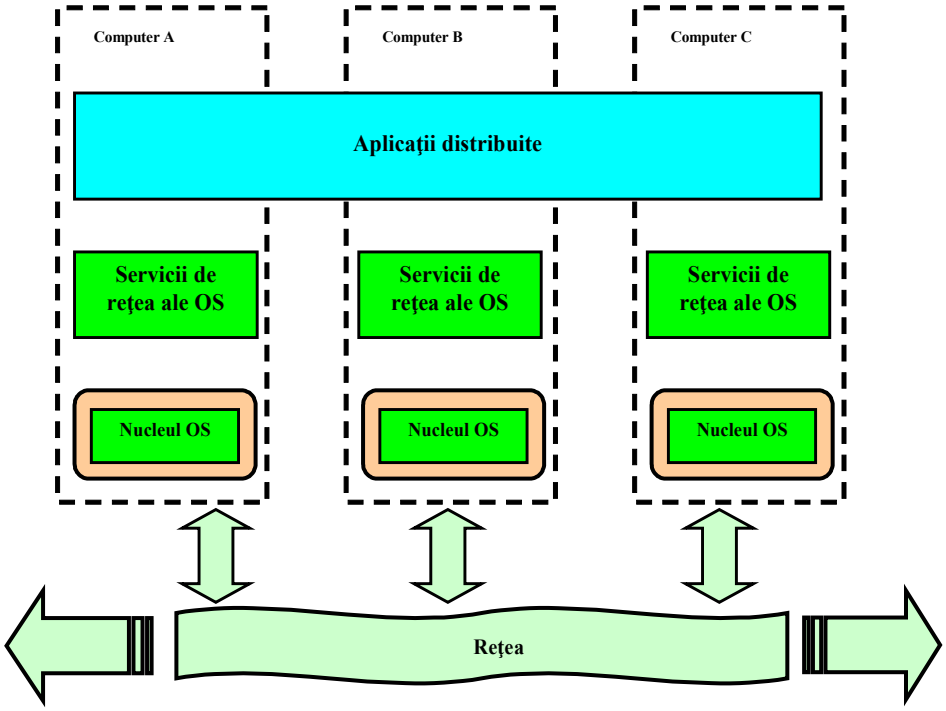

Figura 7.4 – Structura generală a unui sistem de operare în rețea

Mașinile componente și sistemele de operare individuale pot fi diferite, dar ele sunt toate interconectate prin intermediul unui software de rețea. Serviciile de intercomunicare în rețea au însă limitări în ceea ce privește asigurarea nivelului de transparență în utilizarea resurselor și disponibilitatea acestora, cerințe specifice unui sistem distribuit (a se vedea aspectele vizate de nevoia de transparență într-un sistem distribuit, prezentate anterior).
Din acest motiv, nici sistemele de operare distribuite, nici sistemele de operare în rețea nu satisfac cu adevărat cerințele formulate la începutul acestui capitol pentru conceptul de sistem distribuit. Un sistem de operare distribuit nu este menit să administreze o colecție de computere independente, în timp ce un sistem de operare în rețea nu poate oferi imaginea unui singur sistem coerent.
Pentru a beneficia de facilitățile oferite de ambele configurații este necesară introducerea unui nivel software adițional, care să poată fi utilizat într-un sistem de operare în rețea și care să mascheze eterogenitatea componentelor sistemul și oferi, în același timp, transparența necesară în ceea ce privește distribuția acestor componente. Cele mai multe din sistemele distribuite moderne sunt bazate pe construcția acestui nivel adițional de software numit *middleware* ***(Figura 7.5)***.
Pentru a simplifica procesul de dezvoltare și integrare a aplicațiilor distribuite, componenta de *middleware* trebuie să se bazeze pe un anumit model, sau paradigmă, în ceea ce privește distribuția componentelor și comunicarea dintre acestea. Din această perspectivă au existat, istoric, mai multe abordări. Unul dintre modele propune ca fiecare

resursă din sistem să fie tratată ca și un fișier (abordare originar introdusă în UNIX). O extensie a acestei abordări este considerată configurația cu sistem de fișiere distribuit.

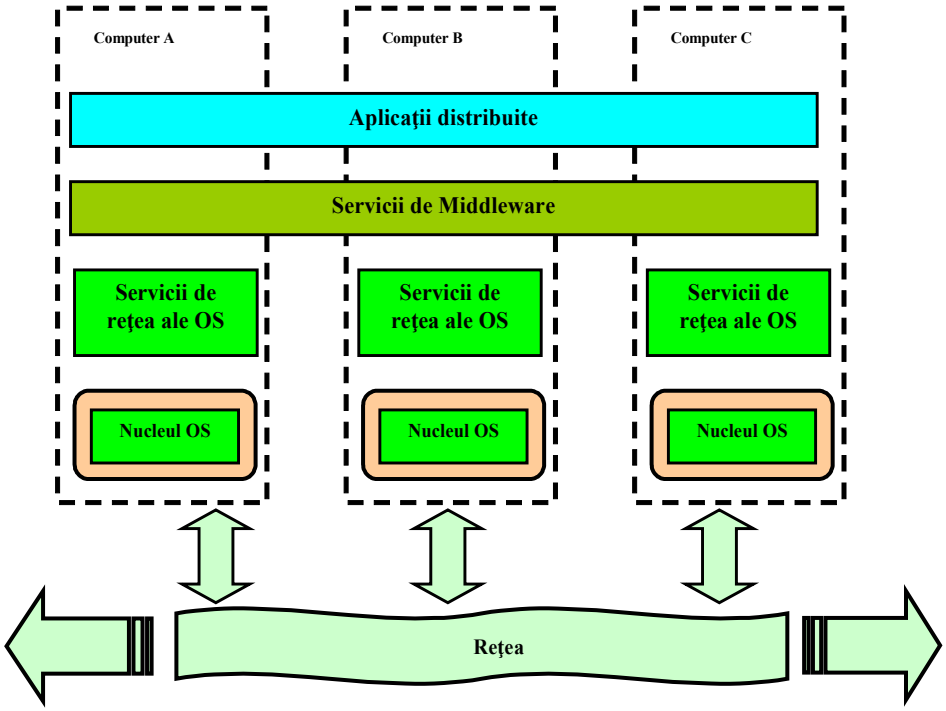

Figura 7.5 – Structura generală a unui sistem distribuit ca și Middleware

O altă abordare a nivelul *middleware* a fost, încă de timpuriu, cea bazată pe apeluri de proceduri la distanță (*Remote Procedure Calls* – RPC). În acest model accentul cade pe ideea de a face transparentă comunicarea în rețea, prin punerea la dispoziția proceselor a unui mecanism prin care li se permite apelul unei proceduri a cărei implementare se află pe o altă mașină (la distanță) în rețea. Când o astfel de procedură este apelată, parametrii acesteia sunt transferați, într-o manieră transparentă, către mașina de la distanță, unde procedura este executată, rezultatele fiind trimise înapoi către mașina apelatoare, respectându-se același grad de transparență al comunicării. Totul apare ca și cum a fost executat un apel local. Procesul care inițiază apelul nesesizând că a avut loc de fapt o comunicare în rețea, poate doar cu excepția unei anume scăderi în performanță.
Mergând mai departe, odată cu paradigma programării obiectuale, au apărut sisteme *middleware* care susțin conceptul de obiecte distribuite. Esența ideii de obiecte distribuite constă în faptul că fiecare obiect implementează o interfață care ascunde astfel detaliile interne de implementare a obiectului pentru utilizatorii acestuia. O interfață constă în metodele pe care obiectul le implementează, nici mai mult și nici mai puțin.

8. Asupra unei arhitecturi de trading orientată pe servicii

De la apariția lor și până în prezent, sistemele electronice de pentru realizarea de tranzacții bursiere au cunoscut o dezvoltare semnificativă, în ceea ce privește complexitatea și rafinamentul serviciilor oferite utilizatorilor. Nevoia de sisteme distribuite care să fie reziliente, mai facil de ajustat pentru a face față unui flux mai mare de date și mai flexibile decât în trecut, a condus la creșterea complexității arhitecturilor avute în vedere. Așa cum am văzut în capitolele anterioare, designul arhitectural al sistemelor de *trading* a evoluat continuu, în pas cu tehnologiile informatice ale momentului. Ideea unei platforme de comunicație care să poată pune în conexiune aplicații scrise în limbaje de programare diferite, pentru platforme diferite și care care rulează într-un mediu de calcul distribuit și eterogen, a constituit una dintre cerințele cele mai stringente ale dezvoltatorilor de aplicații software. Cu toate că interfața pusă la dispoziție de *Java Message Service* (JMS) nu s-a modificat semnificativ de la introducerea ei în 1999, maniera în care un *middleware* orientat pe mesaje este utilizat în practica informatică s-a schimbat dramatic. Sistemele de mesagerie sunt astăzi utilizate pentru a rezolva probleme legate de siguranța livrării datelor între aplicații, probleme legate de capacitatea de ajustare a dimensiunilor sistemelor, în funcție de volumul datelor de prelucrat, dar și probleme specifice diverselor domenii, atât din sfera afacerilor cât și din domeniul educațional, medical etc.

Posibilitatea integrării unor sisteme eterogene este unul din principalele aspecte în care platformele de mesagerie joacă un rol cheie. Un *Message Oriented Middleware* (MOM) oferă abilitatea de procesare a cererilor în mod asincron, dându-le astfel posibilitatea proiectanților și dezvoltatorilor de a furniza soluții care să reducă sau să chiar să elimine strangulările în transferul de date între componentele unui sistem informatic, crescând în acest fel productivitatea utilizatorului final și posibilitatea sistemului, în ansamblul lui, de a face față unui volum de date de prelucrat care poate crește semnificativ de-a lungul perioadei de exploatare. Sintetizând, avantajele oferite de MOM sunt următoarele:

- posibilitatea integrării de componente eterogene, rulând pe platforme diferite, într-un sistem informatic distribuit;
- reducerea strangulărilor în sistem – decuplarea aplicațiilor, prin interschimbul asincron de mesaje, oferă fiecărei componente posibilitatea de a prelucra datele în ritmul propriu, fără a afecta performanțele altor componente ale sistemului; dacă necesarul de prelucrare este mai ridicat pentru anumite componente, acestea pot, în mod dinamic, beneficia de multiple instanțe ale aceluiași proces;
- creșterea capacității de sistemului de a face față unor volume mari de date, prin posibilitatea de a adăuga dinamic componente pentru a prelucra concurențial cozi de mesaje omogene;
- posibilitatea proiectării unor sisteme cu arhitecturi flexibile și agile, care se pot ușor adapta nevoilor concrete de exploatare;
- creșterea productivității utilizatorului final – comunicarea în mod asincron între aplicațiile componente ale unui sistem, îi oferă posibilitatea utilizatorului de a nu aștepta în mod necesar un răspuns imediat (sincron) de la un furnizor de servicii, ci de a-și plasa cererile și apoi continua activitatea până la primirea (asincronă) a rezultatelor cererilor sale de prelucrare.

Maniera generală de interconectare a două aplicații (distribuite) prin intermediul unui MOM (*middleware* orientat pe mesaje) este ilustrată în ***Figura 8.1***.

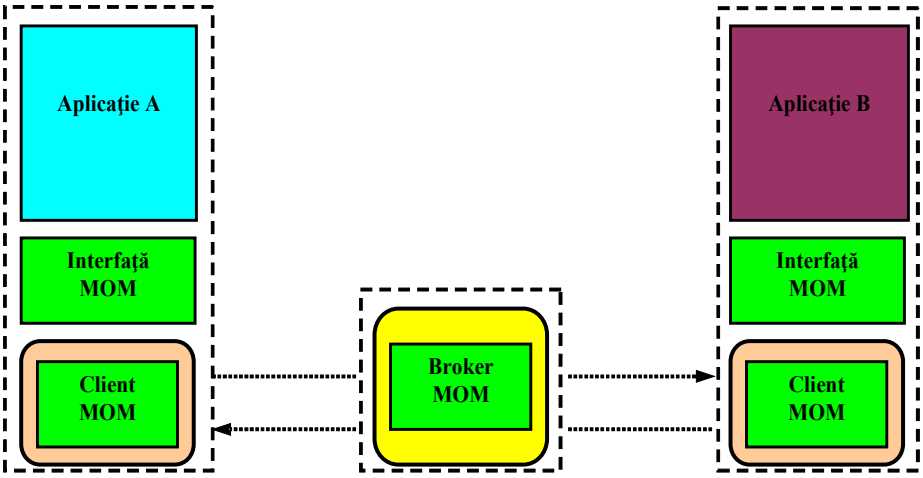

Figura 8.1 – Comunicarea între aplicații prin intermediul unui middleware orientat pe mesaje

Unul dintre conceptele de bază ale unui *middleware* orientat pe mesaje este acela de livrare asincronă a mesajelor între sistemele conectate într-o rețea. Livrarea asincronă a mesajelor presupune că aplicația care trimite un mesaj nu trebuie să aştepte ca respectivul mesaj să fie recepționat sau prelucrat de aplicația căreia i-a fost destinat; aplicația care produce mesajul are libertatea de a trimite mesajul în cauză și continua apoi propriul flux de prelucrare. Mesajele asincrone sunt manipulate ca unități de sine stătătoare, în sensul în care fiecare mesaj conține toate datele de business și cele de stare necesare pentru a putea fi prelucrat din perspectiva logicii aplicației ce se găsește la destinație. În filosofia de mesagerie asincronă, aplicațiile utilizează o interfață (*Application Programing Interface* - API) simplă pentru a construi mesaje, iar aceste mesaje sunt ulterior pasate către platforma de comunicație orientată pe mesaje (MOM), pentru ca aceasta din urmă să le livreze către destinațiile avute în vedere.
Arhitecturile utilizate pentru implementarea MOM pot urmări diferite abordări. De la o arhitectură centralizată, care se bazează pe un server ce realizează rutarea mesajelor, până la arhitecturi descentralizate, în care se distribuie componenta de rutare a mesajelor pe mașinile clienților. De asemenea, protocoalele de comunicație utilizate la nivelul de rețea transport includ TCP/IP, HTTP, SSL și IP multicast.
Este important de menționat că, în contextul utilizării unei platforme de comunicație orientate pe mesaje, termenul de client capătă o semnificație mai largă, în sensul în care toate aplicațiile conectate la platforma de comunicație devin clienți ai acestui *middleware*, chiar dacă din perspectiva logicii funcționale a sistemului informatic, privit în ansamblul său, pot juca rol de client, respectiv de server. Sistemele de comunicație orientate pe mesaje sunt compuse din mesajele aplicațiilor client ale platformei și componenta de server de *middleware* pentru rutarea acestor mesaje. Clienții transmit mesaje către server de mesagerie, mesaje care sunt apoi distribuite de acesta către alți clienți ai sistemului. În concluzie, clientul este o aplicație informatică sau o componentă care utilizează interfața (API) pusă la dispoziție de MOM. Vom prezenta în continuarea acestui capitol interfața *Java Messsage Service* (JMS).

8.1 Arhitecturi MOM centralizate

Sistemele de mesagerie care se bazează pe o arhitectură centralizată folosesc un așa-numit server de mesaje. Serverul de mesaje, care poate fi de asemenea denumit și *router* de mesaje, sau *broker* de mesaje, este responsabil cu livrarea mesajelor între clienții sistemului de mesagerie. Serverul de mesaje realizează, în acest fel, decuplarea clientului care produce mesajul de cel care consumă respectivul mesaj. Clienții intră în contact numai cu serverul de mesaje și nu cu ceilalți clienți ai sistemului, permițând eliminarea sau adăugarea de noi clienți, fără afectarea sistemului ca un întreg. În mod normal, o arhitectură centralizată utilizează o topologie de tip stea. În configurația cea mai simplă, avem de a face cu un broker de mesaje plasat central, la care se conectează toți clienții sistemului. Așa cum se poate observa în ***Figura 8.2*** arhitectura de tip stea oferă un număr minim de conexiuni de rețea, interconectând totodată fiecare componentă a sistemului cu toate celelalte.

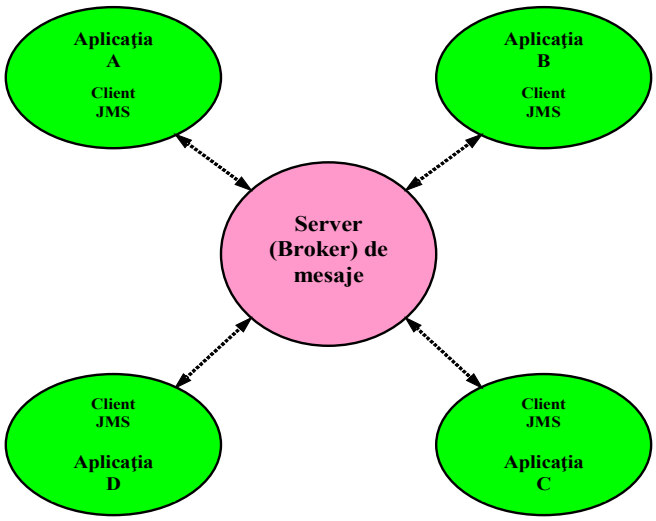

Figura 8.2 – Sistem de mesagerie utilizând o arhitectură centralizată

În practică, serverul central de mesagerie poate fi de fapt un cluster de servere distribuite fizic, dar operând ca o singură entitate la nivel logic. Plecând de la această observație, o arhitectură MOM centralizată poate furniza un model generic de mesagerie pentru sisteme extreme de complexe, oferind un grad înalt de flexibilitate și posibilități facile de redimensionare.

8.2 Arhitecturi MOM descentralizate

Toate arhitecturile descentralizate folosesc la nivel de rețea protocolul *IP multicast*. Un sistem de mesagerie bazat pe *multicasting* nu are un server central pentru livrarea mesajelor. Unele dintre funcționalitățile care ar fi oferite de server sunt, în această arhitectură, încorporate în partea de client (persistare, facilități de lucru cu tranzacții, aspecte legate de siguranța comunicării), în timp ce rutarea mesajelor este delegată către nivelul de rețea, prin folosirea protocolului *IP multicast*. Acest protocol oferă

posibilitatea aplicațiilor de subscrie la unul sau la mai multe grupuri de *multicast*. Fiecare grup de *multicast* folosește o adresă IP de rețea prin care fiecare mesaj ce este primit va fi redistribuit către toți membrii grupului. În acest fel, aplicațiile pot trimite mesaje către o adresă de IP de tip *multicast* și se așteaptă ca nivelul de rețea să asigure distribuirea acestor mesaje în mod corespunzător membrilor care au subscris la acea adresă IP. Rutarea de realizează în mod automat, la nivel de rețea, fără a mai fi necesar un server dedicat pentru face acest lucru (***Figura 8.3***).

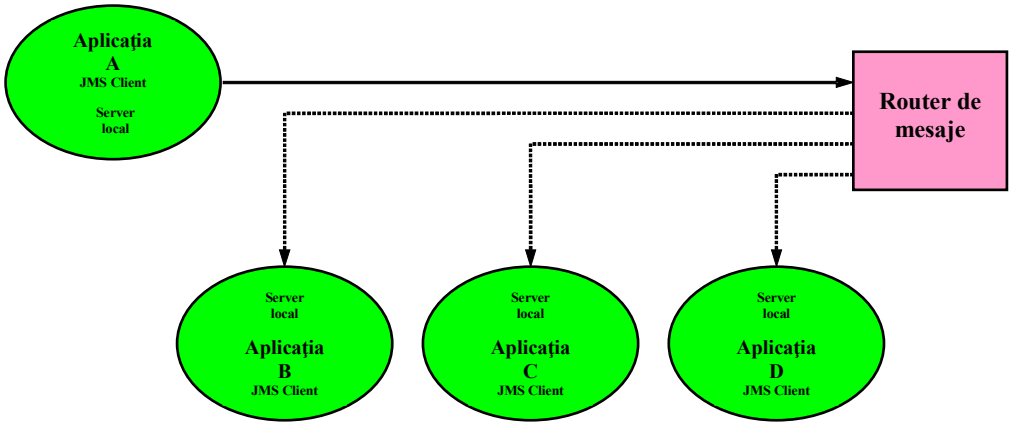

Figura 8.3 – Sistem de mesagerie utilizând o arhitectură IP multicast descentralizată

Cu toate acestea, așa cum am menționat inițial, anumite funcționalități oferite de server vor fi necesar a fi incluse în fiecare client, aspect ce poate conduce la încărcarea clienților și la distribuirea de funcționalități care ar fi mai simplu de gestionat într-un singur loc, cel puțin la nivel logic.

8.3 Arhitecturi MOM hibride – abordarea centralizată ca model la nivel logic

O arhitectură descentralizată implică în mod normal utilizarea protocolului IP *multicast*. O arhitectură centralizată folosește în general protocolul TCP/IP, ca bază pentru realizarea comunicării între diversele componente. O platformă comercială de comunicație, cum ar fi TIBCO *Rendezvous*, poate însă combina cele două abordări. Clienții se pot conecta TCP/IP la un *daemon* care, la rândul lui, comunică cu alte procese *daemon* utilizând grupuri de IP *multicast*. Ambele arhitecturi au avantaje și dezavantaje. Pentru a simplifica prezentarea, fără însă a oferi o perspectivă simplistă asupra subiectului, vom pleca în continuare de la premisa utilizării unei arhitecturi centralizate la nivel logic. În acest context, termenul de *broker* sau server de mesaje se va referi la o componentă arhitecturală care este responsabilă de rutarea și distribuirea mesajelor. Într-o arhitectură centralizată avem de a face cu un server de *middleware* sau cu un cluster de servere. Într-o abordarea descentralizată, serverul se referă la componentele locale de tip server care sunt încorporate în client.

8.4 Modele de mesagerie furnizate de *Java Message Service* (JMS)

Java Message Service (JMS) oferă două tipuri de modele de mesagerie:

- punct la punct (*point-to-point*, sau prescurtat *p2p*);
- publicare și subscriere (*publish-and-subscribe*, sau *pub/sub*).

Aceste modele de mesagerie mai sunt numite și *messaging domains*.
Principial, modelul *pub/sub* presupune publicarea mesajelor de la unul către mai mulți (*broadcast*), în timp ce modelul *p2p* este dedicat transmiterii de mesaje unu la unu (***Figura 8.4***).

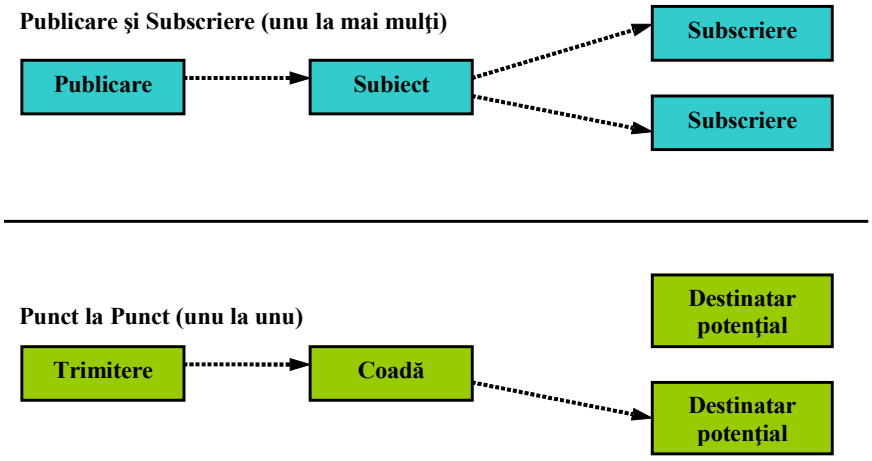

Figura 8.4 – Modelele (domeniile) de mesagerii furnizate de JMS API

Din perspectiva JMS, clienții conectați la platforma de mesagerie se numesc clienți JMS, iar sistemul de mesagerie, cel care furnizează serviciul de distribuire a mesajelor între clienți, se numește JMS *provider*. În general, o aplicație JMS este un sistem informatic care este compus din mai mulți clienți JMS și un server JMS.
În acest context, un client JMS care trimite un mesaj în sistem se numește producător al acelui mesaj, în timp ce un client JMS care primește un mesaj din sistem se numește consumator al respectivului mesaj. Trebuie remarcat faptul că un client JMS poate fi în același timp și producător și consumator de mesaje.

1. Modelul de comunicare punt la punt (p2p)

Acest model permite clienților JMS să trimită și să primească mesaje, atât în manieră sincronă cât și în manieră asincronă, prin intermediul unui canal de comunicație care urmează filosofia de acces a unei cozi. În modelul *p2p* producătorii de mesaje se numesc *senders*, iar consumatorii de mesaje se numesc *receivers*.
În mod tradițional, modelul *p2p* este un model în care mesajele sunt în mod activ cerute din coadă (*pull-based* sau *poll-based*) de către aplicația căreia îi sunt destinate, în loc să fie făcute disponibile către aplicație imediat ce sunt recepționate (*pushed*). Depinzând de implementare, chiar și în modelul *p2p*, serverul de mesaje poate însă să facă disponibile

mesajele din coadă (*push*) aplicației care s-a conectat la aceasta. De exemplu, brokerul OpenMQ face acest lucru.
Una dintre caracteristicile distinctive ale modelului *p2p* este aceea că un mesaj care este trimis către o coadă este primit de către o singură aplicație client și numai una, chiar dacă pot asculta mai multe aplicații client la o aceeași coadă de mesaje.
Acest model oferă două mecanisme de transmitere a mesajelor la nivel de aplicație client:

- trimitere de mesaj fără a aștepta răspuns de confirmare (*fire and forget)*;
- transmisie de tip cerere/răspuns, în manieră asincronă.

Modelul *p2p*, mai ales în cazul mecanismului de cerere/răspuns asincron, tinde să cupleze mai strâns aplicațiile implicate, decât modelul *pub/sub*. De exemplu, un trading GUI poate trimite un ordin către o coadă pentru cumpărarea sau vânzarea unui instrument financiar și apoi să aștepte pentru un răspuns de confirmare, conținând identificatorul unic în sistem al ordinului, generat se serverul de gestiune a ordinelor care prelucrează mesajele trimise către coada respectivă. Pe de altă parte, modelul *p2p* oferă posibilitatea optimizării încărcării sistemului (*load balancing*) în cazul aplicațiilor care consumă și prelucrează mesajele trimise către o coadă. În acest sens se pot lansa în mod dinamic multipli consumatori ai mesajelor unei aceleași cozi, multiple instanțe ale aceleași aplicații, putându-se construi o logică de distribuire optimală a mesajelor cozii între aceste instanțe. Trebuie menționat faptul că specificațiile oferite de JMS nu includ reguli privind distribuirea mesajelor între mai multe aplicații destinatar, însă anumite produse comerciale au ales să implementeze abilități privind echilibrare încărcării consumatorilor.
Față de modelul *pub/sub*, modelul *p2p* oferă posibilitatea aplicațiilor client de a consulta coada de mesaje (*queue browser*) și vizualiza conținutul mesajelor înainte de a le consuma (extrage din coadă).

2. Modelul de comunicare publicare și subscriere (pub/sub)

În modelul *pub/sub*, mesajele sunt publicate către un canal virtual de comunicație numit *topic*. Aplicațiile care produc mesajele trimise către acest canal se numesc aplicații care publică mesaje, în timp ce aplicațiile care consumă mesaje de la un anume *topic* se numesc aplicații care subscriu la acel *topic*. Spre deosebire de modelul *p2p*, mesajele publicate la un *topic* pot fi primite de mai multe aplicații care subscriu la acel *topic*. Această tehnică este uneori numită *broadcast* și reprezintă modalitatea prin care un mesaj este distribuit către toate aplicațiile interesate în recepționarea lui. Fiecare client al serverului de mesagerie care subscrie la un anumit *topic*, va primi o copie a fiecărui mesaj publicat către acel *topic*. În modelul *pub/sub* mesajele sunt puse imediat la dispoziția aplicației client (*pushed-based*) care a subscris la un anumit *topic*, fără ca această să ceară în mod explicit furnizarea mesajelor spre prelucrare. Modelul *pub/sub* oferă un tip de conexiune care nu conduce la o strânsă cuplare a aplicațiile implicate. Clientul serverului de mesagerie care publică mesaje nu are, în general, cunoștință de câți clienți care consumă acele mesaje au subscris la *topic* și ce anume intenționează aceștia să facă cu aceste mesaje. De exemplu, o aplicație care publică date legate de tranzacțiile realizate de către o instituție bursieră, nu trebuie în mod necesar să știe câți clienți au subscris la aceste date și felul în care acești clienți prelucrează datele avute în vedere.

Din perspectiva clienților care subscriu la un *topic*, modelul *pub/sub* oferă posibilitatea realizării a două tipuri de conexiuni:

- conexiune care nu se realizează într-o manieră durabilă, în sensul că aplicația client primește mesajele care sunt publicate la subiectul la care a subscris numai pe durata existenței conexiunii;
- conexiune durabilă, prin care clientul care subscrie la *topic* va primi toate mesajele publicate la acel *topic*, indiferent dacă este conectat sau nu în mod neîntrerupt la serverul de mesagerie; în cazul în care clientul care a subscris de o manieră durabilă la un *topic*, se deconectează de la server sau pierde dintr-un anumit motiv această conexiune, dacă ulterior se reconectează, va primi toate mesajele care au fost publicate la subiectul la care a subscris pe toată durata în care nu a fost conectat la server.

8.5 Interfața pentru proiectarea de aplicații pusă la dispoziție de JMS

JMS este o interfață de comunicare orientată pe mesaje care a fost creată de către Sun Microsystems pe baza specificațiilor JSR-914 (*Java Specification Request*), publicate la începutul anului 2002. JMS nu este un sistem de mesagerie în sine, ci o interfață abstractă împreună cu clasele necesare clienților unui sistem de mesagerie pentru ca aceștia să poată comunica cu sistemul. Utilizând interfața JMS, o aplicație client a unui sistem (server) de mesagerie devine portabilă, din perspectiva serverului concret de mesagerie la care se conectează. Crearea interfeței JMS a fost un efort al întregii industrii software. Obiectivul inițial a fost acela de a furniza un API Java pentru conexiunea la sisteme de mesagerie deja existente la acel moment. Ulterior, scopul inițial al proiectului început de Sun Microsystems s-a lărgit, mergându-se pe ideea oferirii unei alternative viabile de comunicare, prin intermediul unui *middleware* orientat pe mesaje (MOM), la paradigma de comunicare creată prin existența sistemelor bazate pe RPC (*Remote Procedure Call*), precum CORBA și Enterprise JavaBeans.
Interfața JMS este formată din trei componente principale:

- partea generală a interfeței;
- interfața *p2p*;
- interfața *pub/sub*.

În versiunea JMS 1.1, componenta generală a interfeței poate fi folosită pentru trimiterea și recepționarea de mesaje, atât de la o coadă, cât și de la un *topic*. Interfața *p2p* este utilizată exclusiv pentru modelul de mesagerie bazat pe cozi, în timp ce interfața *pub/sub* este destinată utilizării exclusive a modelului de mesagerie bazat pe subiecte (*topics*).
API-ului general este compus din șapte interfețe dedicate trimiterii și recepționării de mesaje:

- **`ConnectionFactory`**
- **`Destination`**
- **`Connection`**
- **`Session`**
- **`Message`**
- **`MessageProducer`**
- **`MessageConsumer`**

Din aceste șapte interfețe generice (***Figura 8.5***), conform specificațiilor JMS, **`ConnectionFactory`** și **`Destination`** trebuie să fie obținute de la serverul de mesagerie, utilizând JNDI (*Java Naming and Directory Interface*).

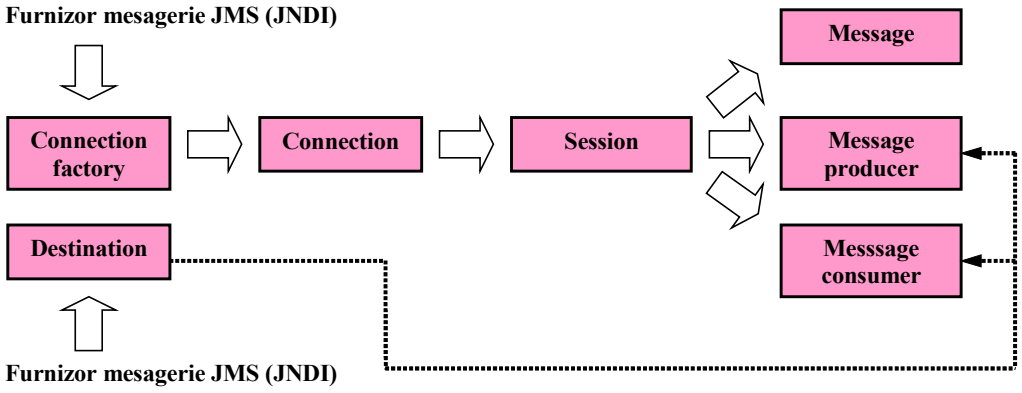

Figura 8.5 – Nucleul de interfețe generice pus la dispoziție de JMS API

Celelalte interfețe sunt create prin diversele metode puse la dispoziție de JMS API. De exemplu, odată ce avem creată o **`ConnectionFactory`**, putem instanția un obiect de tip **`Connection`**. Odată ce avem un obiect de tip **`Connection`**, putem crea unul de tip **`Session`**, iar mai departe, în cadrul instanței **`Session`** putem crea obiecte de tip **`Message`**, **`MessageProducer`** și **`MessageReceiver`** (***Figura 8.5***). În JMS, obiectul **`Session`** deține partea tranzacțională a lucrului cu mesaje și nu obiectul **`Connection`**, așa cum este cazul interfeței JDBC. Aceasta înseamnă că, atunci când utilizează JMS, o aplicație va avea în mod normal numai o singură instanță de tip **`Connection`**, însă poate deține o colecție de obiecte de tip **`Session`**.
Există, de asemenea, în cadrul JMS API, o multitudine de interfețe care sunt destinate gestionării excepțiilor, priorității mesajelor și manierei în care acestea sunt persistate.

8.5.1 JMS API pentru modelul de comunicare punt la punt (p2p)

Interfața JMS pentru modelul de comunicare *p2p* se referă în mod particular la mesageria bazată pe cozi de mesaje. Interfețele utilizate pentru trimiterea și primirea de mesaje în cadrul acestui model sunt următoarele:

- **`QueueConnectionFactory`**
- **`Queue`**
- **`QueueConnection`**
- **`QueueSession`**
- **`Message`**
- **`QueueSender`**
- **`QueueReceiver`**

Întreaga ierarhie de interfețe și maniera prin care acestea interacționează unele cu altele sunt similare modelului general. La nivel sintactic, reflectat până la urmă și în semantica

fiecărei interfețe, conceptul generic de destinație este înlocuit cu cel de coadă (***Figura 8.6***). Aplicațiile care utilizează modelul de mesagerie *p2p* vor folosi în mod normal interfețele bazate pe cozi în locul interfețelor generice.

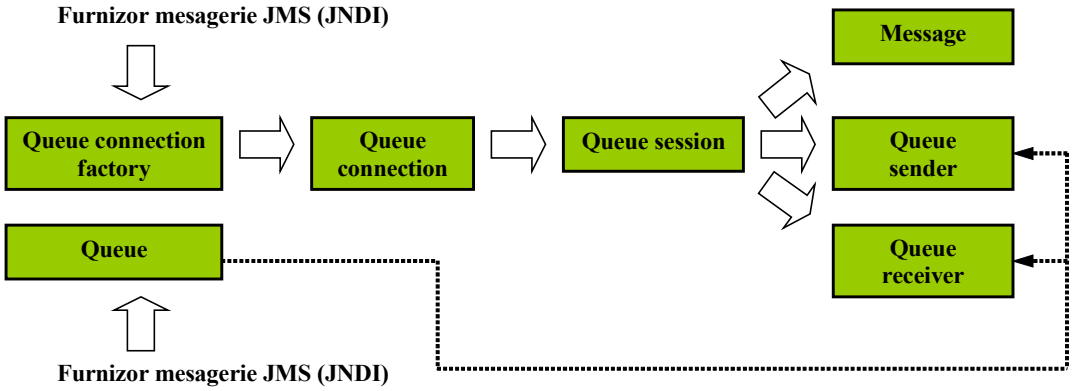

Figura 8.6 – Nucleul JMS de interfețe pentru comunicarea în modelul punct la punct

8.5.2 JMS API pentru modelul de comunicare publicare și subscriere (pub/sub)

Modelul de comunicare pub/sub este oferit de JMS în aceeași manieră standardizată de prezentare a interfețelor. La nivel sintactic cuvântul *Queue* este înlocuit de cuvântul *Topic*. Interfețele pentru modelul *pub/sub* sunt următoarele (a se vedea și ***Figura 8.7***):

- `TopicConnectionFactory`
- `Topic`
- `TopicConnection`
- `TopicSession`
- `Message`
- `TopicPublisher`
- `TopicSubscriber`

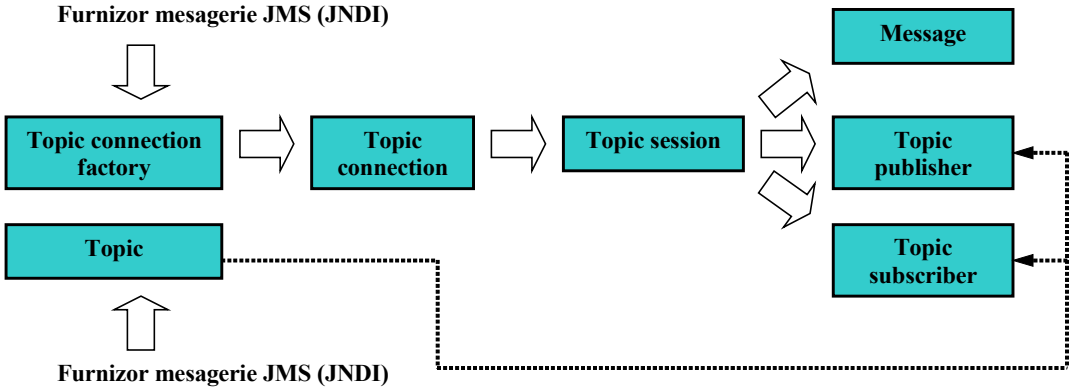

Figura 8.7 - Nucleul JMS de interfețe pentru comunicarea în modelul publicare și subscriere

Singurele excepții, din perpectiva numelui, sunt reprezentate de interfețele **`TopicPublisher`** și **`TopicSubscriber`**. Însă, așa cum subliniam mai devreme, JMS API este foarte intuitiv în ceea ce privește denumirea și utilizarea interfețelor pe care le pune la dispoziție. Din acest punct de vedere, modelul *pub/sub* utilizează *topics* împreună cu *publishers* și *subscribers*, acolo unde modelul *p2p* folosește *queues* împreună cu *senders* și *receivers*. Așa cum este reiese și din ***Figura 8.7***, terminologia și relațiile între interfețele oferite pentru canalizarea mesajelor în cadrul modelului *pub/sub* se mulează fidel pe cele ale API-ul generic oferit de JMS.

8.6 Considerații cu privire la o arhitectură orientată pe servicii

În acest capitol vom încerca să prezentăm câteva scenarii întâlnite în activitatea practică cu privire la problemele pe care un sistem de mesagerie trebuie să le soluționeze în cadrul unei organizații, întreprinderi etc.
O arhitectură orientată pe servicii (*Service Oriented Architecture* - SOA) se referă la un stil arhitectural în care se realizează un proces de abstractizare a serviciilor oferite de un sistem informatic de companie, organizație din mediul de afaceri, pornind de la corespondentul real al acestora, regăsit sub forma departamentelor și serviciilor funcționale din cadrul organizațional avut în vedere. Abordarea de tip SOA a dat naștere unui nou tip de *middleware*, cunoscut sub numele de magistrală de servicii a întreprinderii (*Enterprise Service Bus* – ESB). Așa cum am menționat în capitolele precedente, primele implementări ale unei astfel de magistrale (ESB) au îmbrăcat forma unor servere de mesaje care, în conexiune cu anumite componente incluse la nivel de mesaj, au fost utilizate pentru direcționarea inteligentă a mesajelor, sau pentru transformarea acestora înaintea de a fi livrate către destinație. Un sistem de mesagerie este un mijloc foarte potrivit pentru construirea nivelului de abstractizarea cerut de o abordare de tip SOA. Prin utilizarea mesajelor, componentele care furnizează serviciile informatice, suprapuse peste serviciile oferite în practica organizațională, nu trebuie să fie interesate de aspecte legate de distribuția fizică a altor servicii cu care interacționează, în ce limbaj de programare sunt acestea scrise, sau pe ce platformă rulează; nici măcar nu trebuie să cunoască numele sub care acestea au fost implementate. De asemenea, un sistem fundamentat pe mesaje oferă în cadrul SOA abilități pentru redimensionarea acestuia în funcție de încărcarea la care este supus, precum și mecanisme de monitorizare și control a cererilor care intră în sistem și a răspunsurilor pe care acesta le furnizează mediului extern. Aproape toate sistemele de mesagerie oferite în scop comercial, sau produse sub licență *open source* oferă posibilitatea utilizării interfeței JMS ca protocol generic. O excepție notabilă o reprezintă linia de produse de mesagerie oferită de Microsoft, cum ar fi BizTalk și MSMQ.
În același context, al posibilităților deschise de utilizarea unui MOM, o arhitectură coordonată de evenimente (*Event-Driven Architecture* – EDA) reprezintă un stil arhitectural în care se pleacă de la premisa că orchestrarea proceselor și evenimentelor este o activitate dinamică, extrem de complexă și, în consecință, nefezabil de a fi implementată printr-o singură componentă centrală de coordonare. Atunci când are loc o acțiune în cadrul sistemului, procesul răspunzător de aceasta generează un eveniment ce este trimis întregului sistem, prin care informează asupra acțiunii care a avut loc. Acel eveniment poate la rândul lui declanșa lansarea unui alt proces, care poate, la rândul lui

declanșa lansarea altor procese, fiecare decuplat unul de celălalt. De exemplu, capturarea unei execuții din piața bursieră de către sistemul informatic al unui broker declanșează o multitudine de activități în cadrul sistemului informatic al acestuia: trimiterea execuției către client, transferarea ei către sistemul de gestiune a tranzacțiilor, crearea de către acesta din urmă a unei noi tranzacții, actualizarea portofoliului clientului, generarea unei înregistrări în registrul de confirmări pentru clienți ș.a.m.d.
Cele mai multe companii, prin modelul de dezvoltare urmat, achiziții succesive, fuziuni, migrări, sau pur și simplu prin luarea unor decizii neinspirate, au sfârșit prin a deține o multitudine de platforme eterogene, produse software și instrumente pentru dezvoltarea de aplicații în diverse limbaje de programare. Interacțiunea între aceste platforme poate fi o provocare foarte dificilă, mai ales în condițiile în care standardele continuă să se schimbe și să evolueze. Un MOM poate juca un rol hotărâtor în face posibilă comunicarea între aceste platforme eterogene, fie că este vorba de Java EE și Microsoft .NET, sau Java și C++. În condițiile în care platformele fundamentate pe Java pot utiliza în mod direct API-ul JMS, pentru alte platforme, cum ar fi cele bazate pe .NET sau C++ sunt necesare componente intermediare care să asigure conversia mesajelor dinspre un API nativ către JMS API și viceversa. Majoritate sistemelor de mesagerie oferă astfel de componente (*bridges*), iar acest nivel de jos la care se realizează integrarea diferitelor platforme a dus la lărgirea domeniului la care se manifestă nevoia de integrare.
Majoritatea companiile mature au atât aplicații vechi, perpetuate de-a lungul evoluției companiei, cât și aplicații noi, implementate independent și care pot să nu interopereze în mod necesar cu cele vechi. În multe cazuri, companiile au o stringentă nevoie de a face ca aceste aplicații să poate comunica unele cu altele, pentru a putea partaja și interschimba informații la nivel de companie, ca ansamblu. Integrarea tuturor acestor aplicații la nivel de companie este în general numită *Enterprise Application Integration* (EAI). În acestă direcție au fost dezvoltate diverse soluții, atât de ordin comercial, cât și ca propuneri de implementare venite din interiorul companiei, însă un sistem de mesagerie este componenta centrală a celor mai multe din aceste soluții/propuneri. Dacă, de exemplu în cazul unui sistem de *trading*, subsistemul de gestiune a ordinelor (OMS) este în mod tradițional realizat pe o platformă C/C++, subsistemul pentru gestiunea tranzacțiilor poate foarte eficient fi implementat pe o platformă Java. Interacțiunea dintre aceste două subsisteme, critică în cadrul sistemului de trading luat în ansamblul său, poate fi foarte flexibil realizată prin utilizarea unui *middleware* orientat pe mesaje.
În finalul acestui capitol, vom încerca să comparăm două dintre abordările arhitecturale cele mai utilizate în proiectarea de sisteme informatice distribuite, cu implicații fundamentale în cadrul sistemelor de *trading* și anume:

- abordarea fundamentată pe apelul de proceduri aflate la distanță (*Remote Procedure Call* – RPC);
- abordarea orientată pe mesaje distribuite asincron.

RPC este un termen folosit în mod curent pentru a desemna un model distribuit de calcul, care este folosit atât de platforma Java cât și de cea .NET. Arhitecturile fundamentate pe componente, așa cum este Enterprise JavaBeans, sunt concepute pe baza acestui model. Tehnologiile bazate pe RPC au oferit și continuă să ofere soluții viabile pentru o gamă largă de aplicații. Cu toate acestea, modelul orientat pe mesaje este superior modelului

bazat pe RPC şi vom încerca în continuare să identificăm punctele tari şi cele slabe ale fiecăruia din aceste două modele.

8.6.1 Modelul bazat pe componente interconectate prin RPC

Caracteristicile, facilităţile şi neajunsurile acestui model (***Figura 8.8***) sunt următoarele:

- posibilitatea realizării de aplicaţii ierarhizate pe 3 sau mai multe niveluri; în acest model, nivelul de prezentare (ce constituie primul nivel) comunică utilizând RPC cu nivelul de mijloc, care conţine modelul logic al aplicaţiei (nivelul al doilea) şi care, la rândul lui accesează tot pe baza RPC componenta de gestiune a datelor aplicaţie (care constituie nivelul al treilea). Platformele J2EE furnizată de Sun Microsystems şi Microsoft .NET sunt cele mai moderne exemple ale unei astfel de arhitecturi;
- o arhitectură fundamentată pe RPC încearcă să reproducă comportamentul unui sistem care rulează într-un singur proces; când o procedură aflată la distanţă este invocată, atunci procesul care iniţiază apelul este întrerupt (blocat) până la momentul la care procedura în cauză îşi termină execuţia şi returnează controlul apelatorului; acest model de sincronizare îi permite dezvoltatorului de aplicaţii să vadă într-adevăr sistemul ca rulând într-un singur proces; activităţile sunt realizate în mod secvenţial, asigurându-se faptul că sarcinile sunt îndeplinite într-o ordine prestabilită; natura sincronă a RPC face ca aplicaţia client (cea care efetuează apelul) să fie foarte strâns legată (*tightly coupled*) de server (aplicaţia care satisface apelul); clientul nu îşi poate continua fluxul de procesare până nu primeşte răspuns de la server;
- natura foarte intimă a conexiunilor între componentele unui model bazat pe RPC, creează sisteme în care aplicaţiile sunt puternic interdependente, astfel că eşecul în funcţionare a unei componente poate slăbi funcţionarea altor componente, sau chiar a întregului sistem;

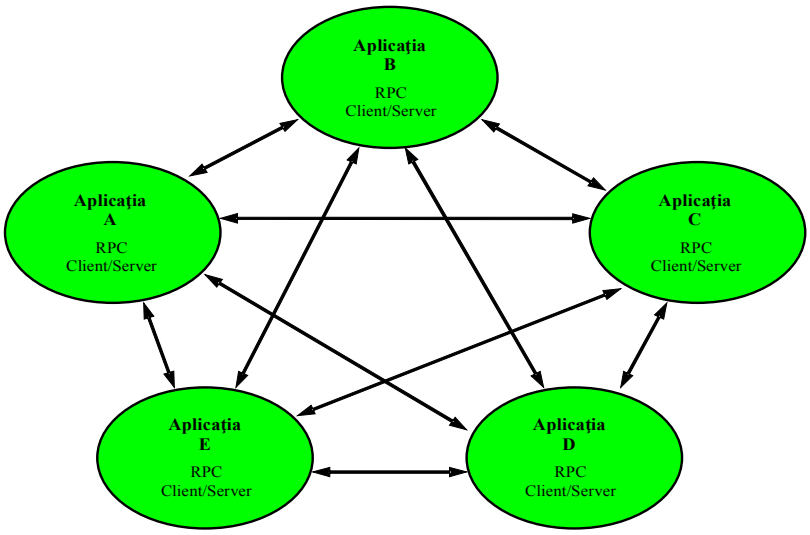

Figura 8.8 – Arhitectură RPC cu aplicaţii tighly coupled

- faptul că maniera de comunicare între aplicații este sincronă, poate să servească foarte eficient diverse scenarii de procesare a datelor, însă face foarte greoaie integrarea pe verticală a sistemelor între ele; într-un scenariu de comunicare sistem-la-sistem, fluxurile de comunicare a datelor pe verticală sunt numeroase și multidirecționale, așa cum se poate vedea în ***Figura 8.8***;
- dacă provocarea de a interconecta aplicații, subsisteme care comunică fiecare cu fiecare nu este suficientă, să ne imaginăm implicațiile care ar rezulta din încercarea de a adăuga un nou subsistem la configurația de ansamblu: este necesară intervenția asupra tuturor celorlalte aplicații/subsisteme cu care noul venit urmează să interacționeze;
- de asemenea, sistemele pot avea disfuncționalități, care generează intervale nefuncționale pentru întregul sistem; momentele de mentenanță și actualizare a interfețelor trebuie să fie planificate; atunci când o componentă este oprită tot sistemul este oprit.

8.6.2 Modelul orientat pe mesaje

Un sistem orientat pe mesaje poate furniza o alternativă viabilă, atunci când tocmai natura sincronă și strânsa interconectare între aplicații devin nepotrivite și neproductive. Caracteristicile și avantajele oferite de un *middleware* orientat pe mesaje (***Figura 8.9***) sunt următoarele:

- problemele legate de disponibilitatea unor subsisteme în cadrul ansamblului, nu constituie un impediment pentru un MOM; conceptul fundamental al unui sistem de mesagerie îl constituie faptul că aplicațiile sunt de presupus să comunice de o manieră asincronă unele cu altele;

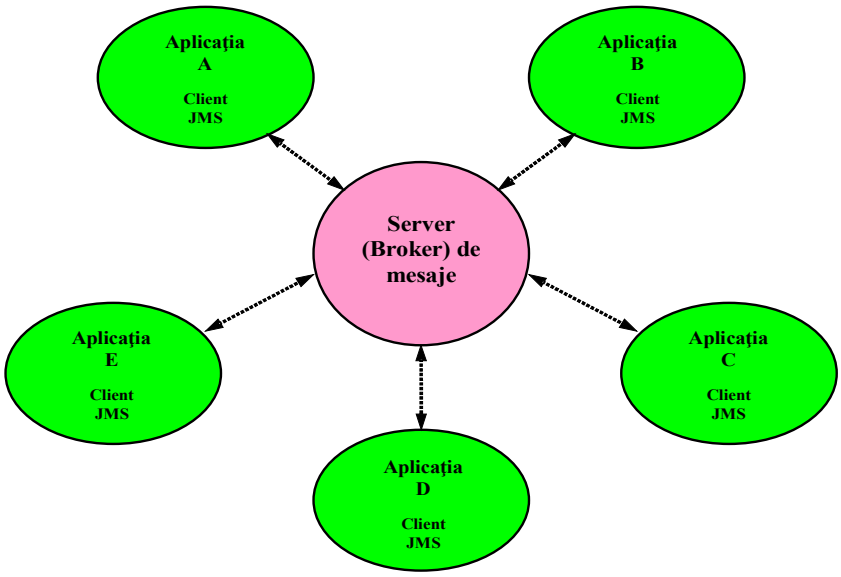

Figura 8.9 – Arhitectură orientată pe mesaje cu aplicații loosely coupled

- codul scris pentru a conecta componentele între ele pleacă de la premisa existenței unui sens unic de transmitere a mesajului, care nu presupune un răspuns imediat din partea aplicației care îl primește; cu alte cuvinte, comparativ cu modelul bazat pe RPC, clientul nu este blocat până la primirea răspunsului de la server; imediat ce un mesaj este trimis de către aplicația client, aceasta poate trece la executarea următoarei sarcini din fluxul de procesare; aceasta este diferența fundamentală dintre modelul sincron bazat pe RPC și modelul asincron furnizat de un sistem de mesagerie; această paradigmă este esențială pentru înțelegerea avantajelor oferite de un MOM;
- aplicațiile/subsistemele unui sistem orientat pe mesaje sunt în mod practic decuplate unele de altele, nu există conexiuni directe de date între aplicații, fluxul de date este canalizat prin serverul de mesaje, sau printr-un cluster de servere de mesagerie; în acest context, eventualele disfuncționalități suferite de o componentă nu afectează funcționarea celorlalte;
- într-un sistem distribuit fizic (chiar geografic), funcționarea necorespunzătoare a unei aplicații sau chiar scoaterea acesteia din funcțiune, reprezintă situații la care este de presupus că ne putem aștepta; interfața JMS pentru un MOM furnizează abilitatea de garantare a livrării mesajelor; prin aceasta se asigură faptul că aplicațiile avute în vedere drept consumatori vor primi în cele din urmă mesajele destinate lor, chiar și în cazul unor căderi parțiale în sistem; livrarea garantată a mesajelor folosește un mecanism de tip *store-and-forward*, ce implică faptul că serverul de mesagerie va persista întotdeauna mesajele pe care le primește, dacă consumatorii avuți în vedere nu sunt disponibili la un moment dat; odată ce aplicațiile consumatoare devin disponibile, acest mecanism le va asigura livrarea tuturor mesajelor pe care nu le-au primit în timpul cât au fost deconectate de la platforma de mesagerie (***Figura 8.10***);

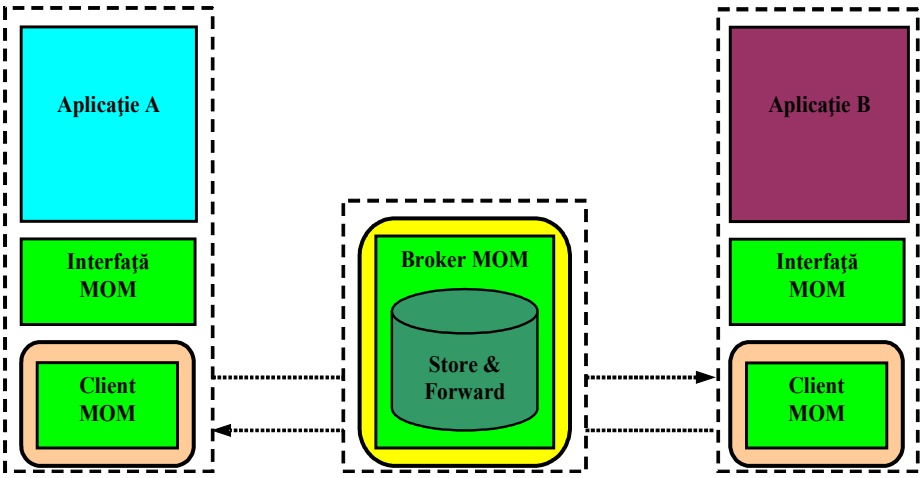

Figura 8.10 – Mecanismul de garantare a livrării mesajelor oferit de serverul de mesagerie

În concluzie, JMS nu este un simplu serviciu coordonat de evenimente. Prin mecanismele de procesare asincronă, de persistare și livrare ulterioară garantată, furnizează funcționalități care oferă aplicațiilor pe care le deservește posibilitatea de a rula într-o manieră continuă, fără necesitatea de a apela de întreruperi pentru

mentenanță. Oferă flexibilitate integrării de noi componente prin furnizarea unor modele de comunicare de tip *pub/sub* și *p2p* într-o manieră asincronă. Datorită transparenței oferite în ceea ce privește localizarea componentelor și a mecanismelor de administrare și control, furnizează premisele unei arhitecturi robuste, orientată pe servicii.