Sisteme ierarhizate şi distribuite

Descrierea CIP a Bibliotecii Naţionale a României
VĂLEAN, HONORIU
Sisteme ierarhizate şi distribuite / Honoriu Vălean. – Bucureşti : Editura Academiei Oamenilor de Ştiinţă din România, 2011
Bibliogr.
Index
ISBN 978-606-8371-26-9

004

Editura Academiei Oamenilor de Ştiinţă din România

Adresa: Splaiul Independenţei, nr. 54, sectorul 5, cod 050094 Bucureşti, România

Redactor: ing. Mihail CĂRUŢAŞU

Documentarist: ing. Ioan BALINT

Coperta: ing. sist. Adrian Nicolae STAN

Copyright © Editura Academiei Oamenilor de Ştiinţă din România, Bucureşti, 2011

Honoriu Valean

Sisteme ierarhizate şi distribuite

Editura Academiei Oamenilor de Ştiinţă din România

Bucureşti

2011

Cuprins

Capitolul 1
Sisteme ierarhizate şi distribuite

Introducere

Teoria sistemelor de conducere ierarhizate şi distribuite nu este nouă. Ea merge în trecut până în anii ’70. Deşi idei legate de folosirea structurilor ierarhizate şi distribuite în calcul şi conducere au fost publicare încă din anii 1960 şi 1970, bazele unei abordări sistematice a sistemelor ierarhizate au fost puse de Mesarovic, Macko şi Tackahara (1970).

La început, studierea sistemelor ierarhizate nu s-a axat pe aspectele practice. Apoi, în literatura de specialitate, teoria controlului ierarhizat a început să piardă din atenție. Datorită studiilor intense din zilele noastre, ea şi-a recâştigat popularitatea.

Sistemele conduse, indiferent de prisma prin care sunt privite – ca instalații tehnologice, ansambluri de procese de producție, sisteme economice – pot fi văzute ca fiind alcătuite din mai multe subsisteme interconectate, eventual dispersate în spațiu.

Ultimii ani ai secolului trecut, ca si începutul secolului 21, au adus în atenție caracteristici noi ale sistemelor mari, industriale sau neindustriale. La ora actuală, o întreprindere trebuie să opereze într-un mediu puternic interconectat în rețea, în condițiile în care cresc preocupările pentru integrarea a numeroase tehnologii diferite, cât şi pentru aspectele de mediu şi sociale.

În consecință, proiectarea sistemelor de conducere trebuie să țină cont de un număr din ce în ce mai mare de factori, ceea ce duce la nevoia de metode şi instrumente de lucru suplimentare. În acelaşi timp, datorită dezvoltărilor tehnologice recente în domeniul calculatoarelor şi al comunicațiilor, au devenit disponibile instrumente eficiente şi o infrastructură tehnică adecvată, care să sprijine proiectarea şi implementarea sistemelor de conducere actuale pentru aplicațiile la scară mare.

Sistemul, ca mulțime de elemente (subsisteme), care interacționează între ele, legate deci în cadrul structurii şi în interacțiunea cu mediul, constituie de asemenea un concept de bază. Însuşirile sistemului, calitățile sale, sunt determinate atât de subsisteme şi relațiile (interacțiunile) interne şi externe, cât şi de relația globală dintre sistem şi subsisteme.

Universul poate fi gândit sau reprezentat, modelat ca un ansamblu de sisteme şi deci de fenomene şi procese legate între ele prin interacțiuni, generându-se unele din altele.

Calitățile, proprietățile sistemelor pot fi asociate cu funcționalitatea, înțelegând prin funcție a unui sistem mulțimea de interacțiuni care realizează

relația sistemului (subsistemului) cu mediul, cu sistemul ierarhic superior. Prin urmare, prin funcționalitate putem înțelege „comportarea în exterior" a sistemului.

Organizarea este asociată cu relațiile care descriu atât continuitatea, stabilitatea sistemelor, repetabilitatea, cât şi tranzițiile între module şi submodule. Aceasta poate fi considerată ca o „comportare în interior" a sistemului.

Prin fenomen se înțelege comportarea dinamică, funcțională, structurală, aleatoare, a unui sistem. Fenomenul se modelează prin şirul de stări sau de evenimente asociate la diferite momente de timp (la momentul ti o stare unică Si).

Prin proces se înțelege comportarea dinamică, funcțională, structurală, aleatoare, a unui subsistem. Acesta se modelează prin şirul de stări sau de evenimente care se obțin conform legii de stare.

Prin urmare, pentru a modela aspectele din realitate, cărora li se asociază concepte (având drept referință categoria de sistem) şi reprezentări de modele (fenomen, proces, structură) este necesară folosirea stării asociată sistemului şi legii de mişcare şi organizare care extinde conceptul de ecuație de stare.

Procesul recurent de cunoaştere a elementelor (subsistemelor) este condiționat de cunoaşterea relațiilor care se repetă. Acestea caracterizează structura, organizarea, sistemul.

Putem considera structura unui sistem ca fiind formată din elemente (subsisteme, E) şi interacțiuni (Ri). Caracterizarea structurii prin cuplul (E, Ri) ne permite să formulăm mai precis ce se înțelege prin sistem ierarhizat. Un sistem ierarhizat este un sistem caracterizat de relația adiacent-subadiacent (de exemplu: un element chimic format din atomi de acelaşi fel este un sistem ierarhizat). Fiecare sistem ierarhizat se poate considera ca o mulțime relativ invariantă de elemente şi interacțiuni în raport cu mulțimea tuturor interacțiunilor posibile din mediul exterior şi, eventual, a unor perturbații, dereglări aleatoare interne, deoarece fiecare sistem este caracterizat de o anumită organizare (structură şi esență) şi de o anumită finalitate.

Deci, prin sistem ierarhizat se înțelege acel sistem de luare a deciziilor de diferite tipuri, organizat pe mai multe niveluri, cărora li se alocă subprobleme specifice, având unele grade de libertate în rezolvare, în condițiile existenței unui mecanism adecvat de coordonare.

Un sistem de producție industrial complex constă, în general, dintr-o multitudine de subsisteme de producție. În interiorul unui subsistem se pot distinge alte subsisteme (de exemplu procese tehnologice) etc.

Deoarece subsistemele concură la realizarea unor produse, între ele pot fi evidențiate anumite interconexiuni (de exemplu: partajarea unor resurse comune). Pe măsură ce subsistemele interconectate sunt mai mici, scad şi dimensiunile elementelor intermediare de legătură. În cadrul structurilor ierarhizate de conducere, fiecărui echipament îi revin sarcini în concordanță cu poziția ocupată în ierarhie.

O organizare ierarhică de succes a unui sistem trebuie să ofere informațiile potrivite subsistemului potrivit la timpul potrivit, indiferent unde este aflat acesta,

ceea ce presupune schimb de informații între subsisteme aflate eventual la diferite niveluri ale ierarhiei.

Prin control se pot înțelege metode de a influența un „obiect" să se comporte în sensul dorit. „Obiectul" poate fi un sistem economic, un proces tehnologic (ex: într-o uzină), un sistem ecologic, un sistem de comunicații, sau poate fi chiar şi un sistem multimedia.

În automatizări a fost necesară dezvoltarea unei structuri ierarhizate pentru a se asigura o fiabilitate sporită, să poată integra aparatura nouă fără a o da la o parte pe cea existentă. În cercetarea operațională, s-a urmărit realizarea unor proceduri de descompunere-coordonare pentru problemele mari de optimizare legate de luarea deciziilor. În informatică, noile dezvoltări tehnologice au impus structurile ierarhizate.

Ideea centrală a teoriei sistemelor ierarhizate (începută încă din anul 1961) a pornit de la structurile existente în ştiința conducerii şi organizării. Astfel, într-o organizație se poate observa o piramidă formată din centre de luare a deciziei pentru rezolvarea unor probleme care variază în complexitate fiind, în general, mai numeroase dar mai simple către bază.

Fiecare dintre aceste probleme se rezolvă, cu o frecvență din ce în ce mai mare spre baza piramidei, ținând seama de anumiți parametri care sunt fixați prin rezolvarea problemelor aflate către vârful piramidei.

Un grup de probleme de decizie prin care se realizează sarcini asemănătoare se constituie într-un nivel al unui sistem ierarhizat.

Sistemele mari (complexe) sunt caracterizate, de regulă, printr-un număr mare de variabile, de neliniarități şi de incertitudini. În mod tradițional, s-a făcut o asociere între descompunerea lor în subsisteme de dimensiuni mai mici, care să poată fi mai uşor conduse si care să fie, eventual, organizate sub o formă ierarhică, şi schimbul de informații intens, dar si cu nevoia elaborării unor mecanisme de coordonare eficiente.

[http://facultate.regielive.ro/proiecte/automatica/sisteme_ierarhizate_de_conducere-193009.html]

Ierarhii

Abordarea ierarhizată a sistemelor distribuite implică descompunerea problemei considerate complexe şi mari într-o mulțime de subprobleme simple. Aceste subrobleme sunt la rândul lor ierarhizate în structuri multi-nivel, având următoarele caracteristici:

- nivelul de vârf conține singură problemă (de obicei de supervizare, coordonare);
- nivelurile inferioare conțin probleme definite de către nivelurile superioare printr-un mecanism de intervenție;
- rezolvarea unei probleme de la un anumit nivel foloseşte informațiile primite printr-un mecanism de reacție doar de la nivelurile inferioare care îi sunt alocate şi subordonate;

- pe fiecare nivel problemele sunt rezolvate în general independent de îndată ce sunt primite informațiile de reacție şi semnalele de intervenție. Există posibilitatea distribuirii sarcinilor pe acelaşi nivel, situație în care rezolvarea problemei implică un mecanism de comunicație şi sincronizare.

În *fig. 1.1* este prezentată structura unui sistem ierarhizat şi distribuit pe 2 niveluri. Nivelul superior reprezintă supervizorul care intervine în deciziile nivelului intermediar. La acest nivel se găsesc elementele de control local CL care iau decizii pe baza informațiilor primite de la subsisteme şi a comenzilor supervizorului. Dacă sarcinile sunt distribuite, există şi un mecanism de comunicație între CL.

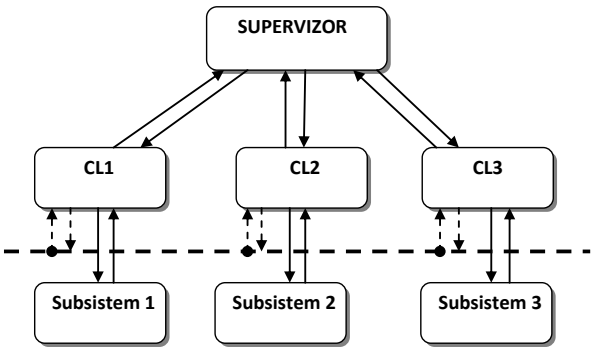

Figura 1.1 Sistem ierahizat distribuit

Arhitecturi pentru sisteme ierarhizate şi distribuite

Arhitectura unui sistem descrie într-o formă generalizată structura sistemului şi interconectarea componentelor acestuia. Aceasta se poate referi la structura hardware, software sau la ambele. Arhitectura unei aplicații hardware sau software trebuie să corespundă specificației acesteia. Proiectarea arhitecturii constă în identificarea şi modelarea structurii folosind informațiile date de cerințele aplicației. Se vor identifica şi realiza toate componentele precum şi modulele de comunicație între acestea.

Diagrame contextuale

Rolul diagramelor contextuale este de a prezenta arhitectura sistemului la un nivel foarte înalt. O structură generală pentru diagrama contextuală a nivelului superior pentru un sistem ierarhizat şi distribuit este prezentată în *fig. 1.2*. O astfel de diagramă este considerată de obicei mult prea generală pentru a satisface cerințele de proiectare. Din această cauză, se preferă descrierea mai amănunțită a componentelor

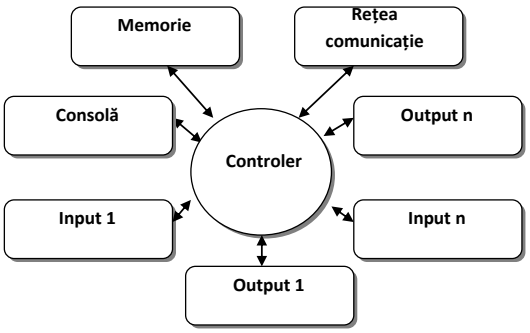

Figura 1.2 Diagrama contextuală de nivel superior

unui sistem ierarhizat distribuit. Un astfel de exemplu este prezentat în *fig. 1.3*, care descrie diagrama contextuală de nivel superior pentru un sistem ierarhizat şi distribuit de control. Această diagramă prezintă arhitectura sistemului incluzând componentele hardware şi software ale sistemului. Operatorii sunt externi sistemului şi îl pot influența prin comenzi transmise, urmărind în acelaşi timp evoluția acestuia.

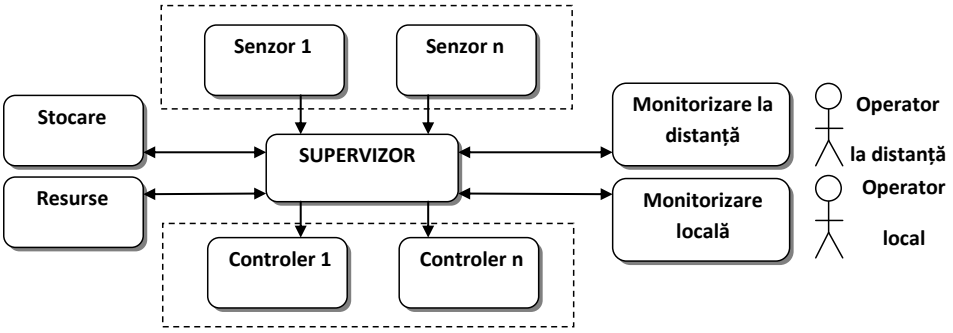

Figura 1.3 Diagrama contextuală de nivel superior

Descompunerea sarcinilor

Aşa cum s-a arătat la paragraful 1.2, rezolvarea sarcinilor în sisteme complexe, în special a celor de control, implică descompunerea lor. Descompunerea se poate face pe vertical, rezultând o ierarhie multistrat de sub-sarcini. O astfel de descompunere a unui sistem de control este prezentată în *fig. 1.4*. În această figură, C1 şi C2 sunt controlere rezultate prin descompunerea sarcinilor iar P procesul. Controlerele pot fi implementate pe aceeaşi structură de calcul sau pe structuri diferite, necesitând asistența unei structuri de comunicație. Funcțiile controlerelor sunt diferite, uzual implementând-se la nivelul inferior sarcini mai simple şi cu o periodicitate de execuție mai mică, iar la nivelul superior sarcini de execuție mai complex şi cu o frecvență de execuție mai mică.

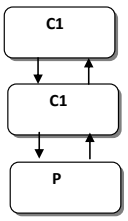

Fig. 1.4

Funcțiile de control pot fi descompuse şi pe orizontală, obținându-se structura din *fig. 1.5*. În acest caz, cele 2 controlere sunt independente, distribuindu-şi sarcinile. Cum ele au obiective diferite este posibil ca în acest caz performanțele sistemului de control să se înrăutățească semnificativ. Deseori pot apare conflicte în deciziile luate de cele 2 controlere datorită obiectivelor locale şi a independenței acestora.

Eliminarea acestui inconvenient se poate face prin adăugarea unei componente cu rol de supervizare (coordonare) notată cu S, ca în *fig. 1.6*. Uneori este nevoie chiar de extinderea acestei structuri pe mai multe niveluri, obținându-se o structură piramidală. În acest caz sistemul este un sistem ierarhizat şi distribuit, controlul fiind de asemenea ierarhizat. Ierarhizarea este vizibilă pe verticală, iar distribuirea

se face simțită la nivelul de control inferior prin distribuirea sarcinilor controlerelor.

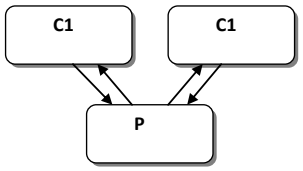

Figura 1.5 Descompunere orizontală

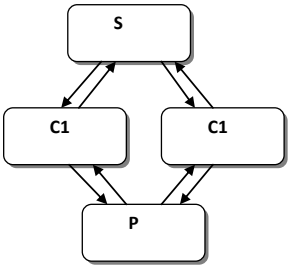

Figura 1.6 Structura ierahizată

Descompunerea planificării activităților

Pentru descompunerea planificării activității pot fi puse în evidență două arhitecturi. Prima este caracteristică sistemelor ierarhizate şi distribuite şi a doua sistemelor distribuite descentralizate.

Arhitectura centralizată

Această structură este caracteristică în general proceselor industriale. Sistemele de calcul înglobate în structura de control sunt folosite pentru gestionarea şi planificarea producției. O astfel de arhitectură este prezentată în *fig. 1.7*. O astfel de arhitectură ierarhizată de control este organizată pe 3 niveluri. Arhitecturile centralizate pot cuprinde în general un număr variabil de niveluri. Sarcinile de conducere sunt distribuite pe nivelurile superioare iar executarea comenzilor transmise de acestea este realizată de nivelurile inferioare. Fluxurile de mesaje in sens descendent conțin comenzi, iar cele în sens ascendent date şi informații de stare. În faza de proiectare trebuie definite complet structura, sarcinile şi formatul mesajelor vehiculate de sistem.

Structura din *fig. 1.7* poate fi implementată pe un singur calculator sau pe o rețea de calculatoare. Într-o astfel de structură toate acțiunile şi comenzile sunt descrise de un calculator central.

O astfel de structură prezintă unele dezavantaje în cazul sistemelor de control. Deoarece nivelul central este responsabil cu coordonarea tuturor

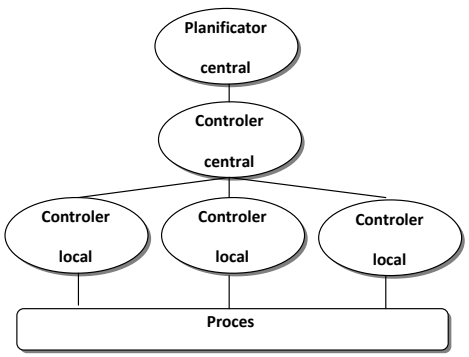

Figura 1.7 Structura centralizată

modulelor, în cazul unor procese complexe se poate ajunge la supraîncărcarea acestuia şi imposibilitatea îndeplinirii tuturor sarcinilor în timp-real. În plus, în cazul apariţiei unor erori la nivelul central, acestea se propagă la toate celelalte niveluri.

Arhitectura heterarhică

Arhitecturile de control distribuit înlătură dezavantajele sistemelor centralizate şi ierarhizate. Arhitectura sistemului trebuie creată luând în considerare specificaţiile, funcţiile de control, informaţiile şi interacţiunile dintre modulele rezultate. Scopul descompunerii acestor sisteme este de a obţine un set de elemente autonome, independente care să aibă capacitatea de a-şi modifica comportamentul pe baza observaţiilor făcute asupra mediului în care evoluează.

Componentele unui sistem de control descentralizat au ca şi criteriu local de performanţă îndeplinirea sarcinilor (rezultate din specificaţii) care au fost planificate iar ca şi criteriu global de performanţă obţinerea unui sistem ale cărui performanţe să se încadreze între limite predefinite.

O astfel de arhitectură este prezentată în fig. 1.8. Astfel de sisteme trebuie să conţină resurse hardware (de prelucrare, de transport, de comunicare, dispozitive periferice, etc.) şi software (baze de date, tampoane de memorie, taskuri, obiecte comune, etc.). Realizarea performanţelor dorite ale sistemului impune utilizarea judicioasă a resurselor şi se obţine prin planificarea lor. În cazul unui sistem distribuit este nevoie de o planificare distribuită. Responsabilitatea planificării este distribuită la nivelul fiecărei entităţi locale, acestea având rolul de a realiza planificările locale ale execuţiei sarcinilor şi alocării resurselor. Aşadar, într-un astfel de sistem, problema planificării este descompusă în problem mai mici şi mai puţin complexe şi este distribuită pe mai multe unităţi de prelucrare. Principalul dezavantaj al unor astfel de sisteme este posibilitatea apariţiei lipsei de coerenţă a sistemului global.

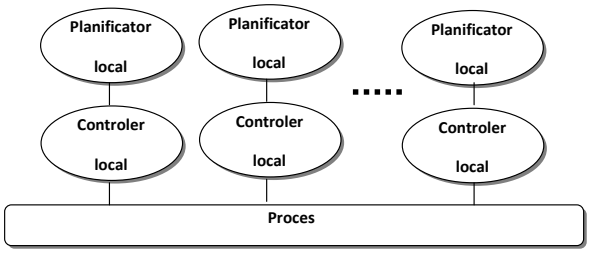

Figura 1.8 Structura heterarhică

În cazul arhitecturilor heterarhice se au în vedere următoarele principii:

- nu există un planificator central;
- componentele locale cu rol de control şi planificare nu au o viziune de ansamblu asupra stării sistemului global;
- fiecare dintre componentele sistemului responsabile cu planificarea pentru luarea deciziilor realizează un compromis între performanţele locale şi cele globale.
- componentele cooperează între ele pentru îndeplinirea criteriilor de performanţă globale conform principiului celor mai puţine angajamente.

Relaţiile de angajare a efectuării unor sarcini pentru alte componente trebuie întârziate cât mai mult posibil şi trebuie terminate cât mai curând posibil.
- entităţile trebuie să coopereze cu alte entităţi oricând este posibil, în scopul rezolvării optime a conflictelor pentru utilizarea resurselor comune.
- entităţile trebuie să respecte planificările oricând se poate;
- nu este nevoie ca entităţile să cunoască planificările altor entităţi. Trebuie să se evite utilizarea unor date globale şi să se minimizeze dependenţele între entităţi;
- entităţilor trebuie să li se impună cât mai puţine constrângeri de proiectare.

Arhitecturi pentru sisteme ierarhizate şi distribuite

O problemă importantă pe care trebuie s-o rezolve proiectantul unui sistem ierarhizat şi distribuit este descrierea lui. Descrierea este rezultatul analizei cerinţelor sistemului luând în considerare toate aspectele funcţionale şi constrângerile impuse.

Specificarea trebuie să fie făcută astfel încât să cuprindă cele mai importante caracteristici ale sistemului. Activitatea de descriere poartă denumirea de specificare, iar rezultatul ei specificaţie.

Specificarea este urmată de analiza specificaţiilor şi verificarea. Scopul acestor activităţi este
- îmbunătăţirea funcţionalităţii sistemului şi creşterea productivităţii activităţilor de proiectare şi implementare;
- garantarea unor factori de calitate, cum ar fi siguranţa, consistenţa, respectarea cerinţelor temporale, etc.;

Specificaţiile cuprind aspecte structurale, funcţionale şi comportamentale. Aspectul structural se referă la descompunerea sistemului în subsisteme şi relaţiile dintre acestea. Cel funcţional se referă la activităţile de transformare pe care trebuie să le realizeze sistemul (echipamentele, componentele hardware, software). Aspectul temporal cuprinde caracteristicile dinamice ale sistemului, cum ar fi reacţia la evenimente interne sau externe, sincrone sau asincrone.

Specificaţia trebuie să se refere atât la componentele hardware şi software ale sistemului, cât şi la procesul condus (în cazul sistemelor de control) sau activitatea operatorului. În caz contrar se va putea verifica numai activitatea specificată. În general, natura diferită a componentelor îngreunează tratarea lor unitară.

Clasificarea tehnicilor de specificare

Există un număr mare de tehnici de specificare. Luând în considerare gradul de formalism utilizat, tehnicile se pot împărţi în formale şi informale.

Tehnicile formale se bazează pe concepte matematice, utilizând metodele acesteia. Tehnicile informale se bazează pe structuri şi limbaje naturale.

Sunt preferate uzual tehnicile din prima categorie, deoarece tehnicile informale pot duce la rezultate care sunt incomplete sau inconsistente şi sunt mai dificil de verificat.

O altă clasificare împarte tehnicile de specificare în descriptive, operaționale şi duale.

Tehnicile operaționale folosesc noțiuni de stare şi tranziție pentru descrierea sistemelor. Ele sunt potrivite pentru descrierea aspectelor comportamentale. Pot fi utilizate pentru implementarea directă a modelului sistemului descris. În general, nu pot fi utilizate pentru verificarea proprietăților comportamentale cu sau fără simularea sistemului.

Tehnicile descriptive folosesc notații şi concepte matematice şi produc specificații bazate pe ecuații logice sau algebrice. Acestea pot fi verificate dacă sunt complete sau consistente utilizând proprietăți sau reguli de deducție.

Tehnicile duale folosesc atât posibilități operaționale cât şi descriptive. Problema principală a tehnicilor duale constă în definirea unui mod de trecere de la noțiunile descriptive la cele operaționale şi invers.

Verificarea

Utilizând specificațiile unui sistem se poate efectua verificarea acestuia dacă sunt:

- consistente – nu există greşeli în culegerea şi înregistrarea datelor sau cunoştințelor;
- corecte – nu există cerințe greşite sau în conflict între ele;
- complete – nu există omisiuni.

Verificarea are în vedere specificația cerințelor sistemului care cuprinde:

- cerințe funcționale – ce face sistemul, adică transformările şi activitățile;
- cerințe temporale – când, până când trebuie să facă;
- cerințe nefuncționale – ce atribute are sistemul;
- cerințe de dezvoltare – cum urmează să fie construit.

Verificarea consistenței specificațiilor: în cadrul acesteia se urmăreşte dacă există ieşiri din unele componente declarate ca intrări în alte componente şi care nu sunt compatibile între ele.

Verificarea corectitudinii specificațiilor: este analizată fiecare declarație în parte verificându-se dacă afirmația poate fi realizată practic sau corespunde realității. De exemplu în cazul unei declarații de tipul

Se umple rezervorul în 30 minute

se verifică dacă debitul furnizat pentru umplerea rezervorului poate îndeplini cerința specificată.

Verificarea completitudinii specificațiilor: se urmăreşte dacă există:

- intrări lipsă – se verifică dacă există activități care au definite intrări pentru care nu sunt descrise mărimile sau variabilele efectiv aplicate;
- ieşiri lipsă – se verifică dacă există activități care au definite ieşiri pentru a satisface structura şi modul de lucru al sistemului, dar sunt specificate mai puține ieşiri în specificațiile sistemului;

- ieşiri neutilizate – se verifică dacă au fost definite operații ce au ieşiri care nu sunt utilizate nicăieri în sistem;
- intrări redundante – se verifică dacă sunt intrări specificate în cerințele sistemului dar nu au fost declarate operații care să le folosească;
- operații lipsă – se verifică dacă există definite intrări şi sunt necesare ieşiri dar nu sunt definite activitățile corespunzătoare care să le utilizeze, respectiv sa le producă.

Activitatea care urmează verificării este validarea. Ea constă în urmărirea existenței unor posibile interblocaje, a siguranței precum şi precum şi respectării constrângerilor temporale.

Tehnici de proiectare, verificare şi validare

În cele ce urmează, vor fi amintite principalele tehnici de proiectare, verificare şi validare pentru sisteme ierarhizate şi distribuite. În acest subcapitol vor fi doar amintite, descrierea lor sistematică putând fi găsită în bibliografie.

Principalele tehnici utilizate de proiectanții acestor sisteme sunt:

- Algebra proceselor;
- Diagrame contextuale;
- Diagrame ale legăturilor dintre subsisteme;
- Diagrame ale fluxurilor de date;
- Diagrame ale structurii de procese şi aplicații;
- Diagrame ale dependenței dintre evenimente;
- Maşini de stare extinse;
- Rețele Petri.

[T. Leția, M. Hulea. Sisteme de control distribuit. Ed. Mediamira, 2005]
[T. Leția. Sisteme de timp-real. Ed. Albastră. 2000]

Capitolul 2

Implementarea în java a sistemelor ierarhizate şi distribuite

Odată cu dezvoltarea infrastructurii de comunicaţie, cu ieftinirea serviciilor şi cu scăderea semnificativă a timpului de răspuns, un mecanism din ce în ce mai utilizat pentru implementarea sistemelor distribuite este bazat pe tehnologia web. În capitolele următoare vor fi prezentate tehnici de implementare a sistemelor ierarhizate şi distribuite în cele mai utilizate limbaje. Vom începe cu limbajul Java.

Există o mare varietate de topologii de modele software şi hardware care pot fi utilizate pentru implementarea unui sistem ierarhizat şi distribuit. Câteva dintre acestea sunt:

- modele de timp mainframe – în acest model, întreaga logică a sistemului este concentrată pe un singur calculator. Procesul şi utilizatorul interacţionează cu acest calculator prin intermediul terminalelor. Acestea preiau informaţiile şi comenzile şi le transmit centrului. Răspunsurile calculatorului central sunt furnizate tot prin intermediul terminalelor;
- modele de tip file-sharing – iniţial, sistemele distribuite de calcul erau bazate pe arhitecturi construite prin partajarea fişierelor. Acestea erau descărcate de pe servere pe staţiile de lucru, iar activităţile erau executate pe staţii. Este un model aplicabil doar dacă traficul de date nu este foarte mare;
- modelul client/server – ca urmare a limitărilor introduse de modelul file-sharing, a apărut şi a fost dezvoltat modelul client-server. Această abordare presupune existenţa unui server software care realizează prelucrări de date în numele clientului, returnându-i acestuia rezultate. În principiu, bazat pe acest model se pot realiza implementări ierarhizate şi distribuite, putând implementa descompuneri ale unor probleme complexe în probleme mai simple. Un astfel de exemplu este prezentat în *fig. 2.1*; Serverul central funcţionează pe post de supervizor. La nivelul 2 sunt implementate aplicaţiile de control local, care preiau comenzile serverului central şi generează comenzi spre clienţii distribuiţi. Gradul de complexitate al unui astfel de model este mare, apărând o serie de

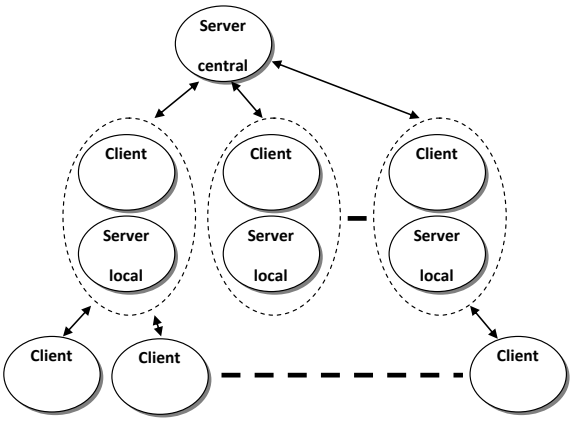

Figura 2.1 Sistem software ierarhizat şi distribuit

aspecte legate de securitate, partajarea resurselor, sincronizarea şi comunicarea între module, etc.

- modelul peer-to-peer – se bazează pe puterea de calcul a clienților şi lărgimea de bandă a mediului de comunicație. În cadrul acestui model nu există noțiunile de client şi server (*fig. 2.2*). Acest tip de rețele sunt uzual folosite pentru accesarea comună a datelor din rețea. Unele aplicații distribuite, cum ar fi sistemele multiagent respectă acest model;

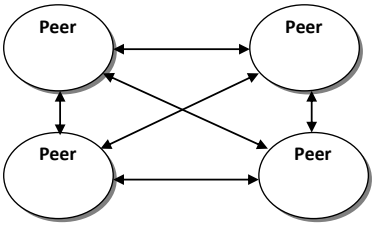

Figura 2.2 Peer-to-peer

- modelul codurilor mobile – are la bază principiul transmiterii la distanță a codurilor executabile pentru a fi executate pe maşinile client (*fig. 2.3*). Una dintre cele mai cunoscute tehnologii care are la bază acest principiu este tehnologia agenților software, despre care se va discuta într-un capitol separat. Tehnologia codurilor mobile cuprinde următoarele tipuri (în java)
  - applet – aplicații descărcabile
  - servlets – servicii încărcabile
  - deglets – taskuri delegate
  - netlets – taskuri autonome
  - piglets – coduri mobile malițioase

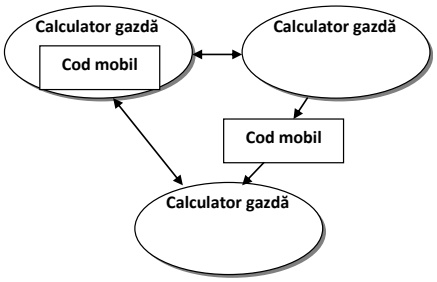

Figura 2.3. Cod mobil

În continuare vom vedea cum pot fi implementate unele din aceste modele.

Implementarea aplicațiilor client-server

Pentru realizarea unor programe care comunică în rețea se utilizează clasele din pachetul java.net. Câteva din clasele acestui pachet sunt prezentate în tabelul 2.1

Java oferă două abordări diferite pentru realizarea de programe de rețea, asociate cu clasele

- Socket, DatagramSocket şi ServereSocket
- URL, URLEncoder şi URLConnection

Utilizarea socket-urilor implică o programare de nivelul inferior pentru comunicarea de date între 2 entități. Ca şi principiu, acest mecanism permite realizarea comunicării în mod ful-duplex între server şi client. Comunicarea se face prin fluxuri de octeți.

Tabelul 2.1

Clasele pachetului java.net

Clasa	Scop
URL	Reprezintă un URL
URLConnection	Returnează conținutul adresat de obiectele URL
Socket	Creează un socket TCP/IP
ServerSocket	Creează un socket TCP/IP pentru un server
DatagramSocket	Creează un soclu UDP
Datagram Packet	Implementează o datagramă trimisă printr-un obiect DatagramSocket
InetAddress	Reprezintă numele şi IP-ul unui calculator din rețea

Clasele din pachetul java.net oferă doar suportul necesar implementării mecanismelor de comunicare între aplicații. Rolul programatorului este acela de a implementa protocoalele pe baza cărora aplicațiile schimbă mesaje.

Un socket reprezintă un punct de conexiune într-o rețea TCP/IP. Când 2 calculatoare din rețea doresc să comunice, fiecare îşi creează un socket. Unul din programe (serverul) va deschide un socket de ascultare şi va aştepta o cerere de conectare de la celălalt calculator. Când clientul se conectează la server, schimbul de informații poate începe. Pentru a stabili o conexiune, clientul trebuie să cunoască adresa destinației şi portul atribuit socket-ului.

Implementarea unei aplicații client-server

Pentru implementarea unei aplicații client-server se utilizează clasele ServerSocket şi Socket. Aceste clase reprezintă interfața prin care programatorul are acces la mecanismele de comunicare prin protocolul TCP/IP. Detaliile de implementare ale protocoalelor sunt încapsulate în interiorul claselor şi sunt transparente pentru utilizator.

Serverul va trebui să creeze un socket, să deschidă un port şi să aştepte conexiuni. În acest sens va instanția un obiect de clasă ServerSocket. La crearea acestui obiect se specifică şi portul pe care se realizează „ascultarea”. Inițializarea ascultării portului se face prin apelul metodei accept (). La momentul execuției acestei metode procesul server se va bloca până la apariția unei cereri de conectare. Metoda accept () returnează un obiect de tip Socket pe care se va face în continuare comunicația, obiectul ServerSocket continuând să „asculte” mediul. Secvența de program care realizează aceste acțiuni este

```
ServerSocket ss=new ServerSocket (2000);
Socket socket=ss.accept ();
```

La rândul său clientul pentru a se conecta la server va trebui să creeze un obiect Socket care va primi ca şi parametri adresa serverului şi portul pe care acesta ascultă. O adresă este reprezentată sub forma unui obiect InetAddress.

```
InetAddress servadr=InetAddress.getByName („localhost”);
Socket socket=new Socket (servadr,2000);
```

Atât la nivelul serverului cât şi la nivelul clientului, prin crearea obiectelor de tip Socket se vor obține fluxuri de date care pot fi scrise sau citite. Metodele utilizate pentru aceste acțiuni sunt getInputStream () şi getOutputStream ().

```
BufferedReader in=new BufferedReader (new InputStreamReader (socket.getInputStream ()));
PrintWriter out=new PrintWriter (new BufferedWriter (new
OutputStreamWriter (socket.getOutputStream ()),true);
```

În secvența anterioară este exemplificat un modul în are se pot constitui fluxuri de intrare-ieşire orientate pe şiruri de caractere. Pentru utilizarea acestor fluxuri se va importa pachetul java.io. Metodele getInputStream () şi getOutputStream () oferă referințe spre fluxuri intrare-ieşire orientate pe octet. Dacă se doreşte lucrul cu fluxuri orientate spre caractere se utilizează clasele de conversie InputStreamReader şi OutputStreamWriter.

La finalizarea transferului de date între client şi server se recomandă închiderea socket-ului:

```
Socket.close ();
```

Servere multifir

Aplicațiile server pot fi: monofilare, secvențiale şi multifilare. Aplicațiile multifilare lucrează în mod concurent, având facilitatea de a deservi simultan mai mulți clienți. Aplicațiile server din *fig. 2.1* sunt în mod obligatoriu multifilare.

Un exemplu de server multifilar este prezentat mai jos.

```
import java.io.*;
import java.net.*;
public class ServerMultifir
{
    public static final int PORT=2000;
    void startServer ()
    {
        ServerSocket ss=null;
        try
        {
            ss=new ServerSocket (PORT);
            while (true)
            {
                Socket socket=ss.accept ();
                new TratareClient (socket);
            }
        }
        catch (IOException ex)
        {
            System.err.println („Eroare:”+ex.getMessage ());
        }
        finally
        {
            try{ss.close ();} catch (IOException ex2){}
        }
    }
```

```
	public static void main (String args[])
	{
		ServerMultifir smf=new ServerMultifir ();
		mf.startServer ();
	}
}

class TratareClient extends Thread
{
	private Socket socket;
	private BufferedReader in;
	private PrintWriter out;
	TratareClient (Socket socket) throws IOException
	{
		this.socket=socket;
		in= new BufferedReader (new InputStreamReader (socket.getInputStream ()));
		out=new PrintWriter (new
			BufferedWriter (new OutputStreamWriter (socket.getOutputStream ())));
	}

	public void run ()
	{
		try
		{
			while (true)
			{
				String str=in.readLine ();
				if (str.equals („END”) break;
				System.out.println („Echoing”+str);
				out.println (str);
			}
			System.out.println („closing...”);
		}
		catch (IOException e) {System.err.println („IOException”);}
		finally
		{
			try
			{
				socket.close ();
			}
			catch (IOException e) {System.err.println („Socket not closed”);}
		}
	}
}
```

UDP. Comunicarea prin datagrame

Transmisia datelor prin TCP este sigură şi confirmată. Din acest motiv, datele nu pot fi pierdute, dar lărgimea de bandă a canalului este utilizată neoptim.

Când datele sunt transmise prin UDP ele nu sunt confirmate de destinație. Recepționarea lor nu este garantată și nici ordinea în care ele sosesc. Dar viteza de comunicație creşte simțitor. Există situații în care este mai important ca datele să

fie transmise cu viteză mare decât ajungerea lor în proporție de 100% la destinație. În aceste situații se foloseşte protocolul UDP.

În java, pentru implementarea protocolului UDP sunt utilizate clasele DatagramPacket si DatagramSocket. Spre deosebire de TCP, nu există noțiunea de ServerSocket. Atât serverul cât şi clientul utilizează DatagramSoket pentru realizarea conexiunii şi respectiv DatagramPacket pentru transmisia şi recepția datelor.

Un exemplu de utilizare a protocolului UDP este un server de timp care utilizează datagrame pentru trimiterea la cerere a datei curente către clienți. La nivelul serverului se va crea un obiect DatagramSocket pe care serverul va începe ascultarea.

```
DatagramSocket socket=new DatagramSocket (2000);
```

Se construieşte un obiect DatagramPacket care este utilizat de către server pentru recepția datelor de la client. Odată construit acest obiect, serverul va începe ascultarea pe portul 2000 prin apelarea metodei receive ().

```
byte[] buf=new byte[256];
DatagramPacket packet=new DatagramPacket (buf.length);
socket.receive (packet);
```

În momentul în care un client doreşte să apeleze serviciile serverului, acesta va trimite un pachet către server. Serverul citeşte din pachet adresa şi portul clientului şi îi trimite informația cerută.

```
InetAddress address=packet.getAddress ();
int port=packet.getPort ();
buf= ( (new Date ()).toString ()).getBytes ();
packet=new DatagramPacket (buf, buf.length, address, port);
socket.send (packet);
```

Un client pentru a se conecta la server trebuie să creeze un obiect DatagramSocket şi să trimită un pachet către server. Clientul nu trebuie să specifice nici un port la crearea obiectului, întrucât acest port este creat automat.

Obiecte distribuite

Noțiunea de obiecte distribuite implică localizarea obiectelor pe calculatoare diferite, ceea ce implică următoarele probleme:

- un obiect care trebuie să apeleze o metodă dintr-un alt obiect trebuie să dețină adresa lui;
- eliminarea obiectelor care şi-au terminat durata de viață este mai complexă;
- problema operatorilor logici aplicați obiectelor distribuite este mai complexă;

Cele mai cunoscute metode de apelare a metodelor la distanță sunt

- Remote Method Invocation (RMI) în java;
- Remote Procedure Call (RPC) în C;
- Services în .net;

Pentru identificarea obiectelor distribuite se utilizează identificatori distribuiți (Remote Object Identifier), ca în exemplul următor:

ROID=endpoint (Java Virtual Machine)+identificator (ObjID)

Identificatorii sunt rezolvați de către componente aflate în calculatoarele care implementează obiectele distribuite pentru a face accesul transparent la obiecte. În fiecare calculator care în care există obiecte care trebuie să apeleze metode la distanță există câte un (server) proxy pentru fiecare obiect care are metode de invocat. Obiectul apelant nu apelează direct metoda obiectului la distanță, ci trimite mesajul de apel proxy-ului local care îl transmite mai departe unui skeleoton corespunzător obiectului cu metode de invocat la distanță. Skeletonul se află în acelaşi calculator cu obiectul care are metode de invocat la distanță, iar în calculatorul obiectului apelant există un stub al metodelor de invocat la distanță. Obiectul cu metode apelate poate la rândul lui să apeleze pentru efectuare serviciilor alte metode din alte obiecte. Schematic, acest mecanism este prezentat în *fig. 2.4*.

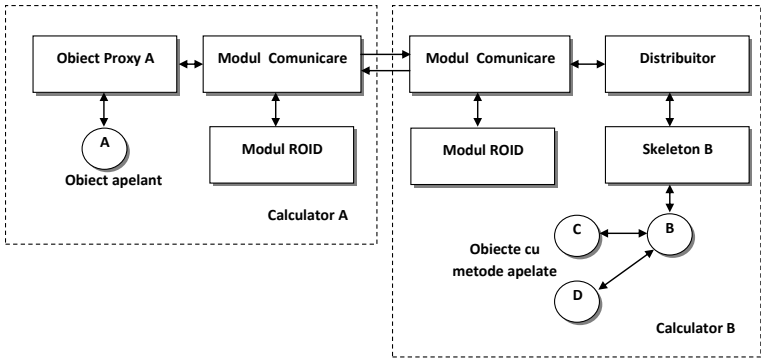

Figura 2.4 Comunicația între obiecte distribuite

Obiectele schimbă mesaje prin intermediul unor module de comunicare. Există module ROID care trebuie să recunoască identificatorii obiectelor pentru ca modulele de comunicare să poată determina cui să transmită apelurile metodelor. Aceste module ROID sunt utilizate la gestiunea proxy-urilor.

Nu este nevoie de distribuitori şi schelete care utilizează facilitatea reflection. De exemplu clasa Method are metoda Object invoke (Object obj, Object args[]) care invocă metoda care stă la baza obiectului Method din obiectul specificat obj cu parametrii dați de args. Parametrii sunt wrapați automat pentru a se potrivi cu parametrii formali primitivi. Dacă metoda care stă la bază este statică, se ignoră argumentul obj.

Trebuie să se redefinească semantica transferului de argumente pentru RMI din limbajele orientate pe obiect. În primul caz se permitea transmiterea de parametri ca referințe ale unor obiecte care puteau conține rezultatele prelucrării la ieşirea din metode. Acest procedeu este mai dificil de aplicat în cazul obiectelor distribuite.

În java RMI dacă se defineşte un parametru ca o interfață îndepărtată, argumentul sau rezultatul este transferat prin ROID. Obiectele care nu sunt îndepărtate pot fi transferate prin valoarea lor dacă nu sunt serializabile.

În concluzie, prin RMI poate fi accesat orice obiect. Diferența dintre obiectele îndepărtate şi cele locale este dată de interfețe.

Serializarea şi transmiterea obiectelor în alte calculatoare permit instanțierea unor obiecte în calculatoarele destinație şi executarea unor metode acolo. Acestea pot fi utilizate pentru echilibrarea încărcării calculatoarelor în sisteme distribuite.

Limbajul java permite legarea dinamică locală, iar această facilitate poate fi extinsă şi pentru implementarea distribuită. Colectarea obiectelor reziduale rezolvată local de garbage collector este mai complicată în aplicațiile distribuite datorită necesității informării componentelor din mediul distribuit dacă mai trebuie utilizate obiecte în viitor. Un colector de reziduuri distribuit realizat cu ROID trebuie să realizeze următoarele:

- să țină evidența numărului numărului de stații îndepărtate înregistrate ROID pentru fiecare obiect local (să mențină o tabelă de evidență a utilizării obiectelor);
- să informeze celelalte module ROID despre generarea sau ştergerea din ROID a obiectelor lor locale (prin utilizarea unor metode tip addRef () şi removeRef ());
- să colecteze reziduurile obiectelor care nu mai sunt referite nici local nici îndepărtat;
- să realizeze numărarea referințelor în rețelele nefiabile (cu metode de tip addROID () şi removeROID ());

Execuția sincronă şi asincronă a metodelor

O problemă importantă este determinarea statutului de activ sau pasiv pentru obiectul îndepărtat. În cazul unui obiect pasiv problema apelului unei metode îndepărtate este simplu de realizat. Obiectul aşteaptă să fie apelat şi execută sincron ceea ce i s-a cerut.

Problema este mai complicată în cazul în care obiectul cu metode de apelat este activ şi se doreşte ca acesta să-şi întrerupă activitatea (în mod asincron) pe care o desfăşoară şi s-o realizeze pe cea specificată de apelant. Acest transfer asincron al execuției este mai complicat deoarece implică întreruperea execuției curente şi înlocuirea ei cu cea cerută.

Implementarea obiectelor distribuite folosind RMI

Pentru aplicația server se defineşte interfața pe care o implementează obiectele distribuite. Această interfață va conține metodele ce vor putea fi apelate la distanță prin intermediul mecanismelor RMI. Interfața va trebui să extindă interfața RemoteInterface. Metodele definite în cadrul acestei interfețe vor trebui să aibă adăugata clauza throws RemoteException.

Se construieşte clasa care implementează interfaţa definită la pasul anterior. Această clasă trebuie să extindă clasa UnicastRemoteObject.

Pe baza clasei definite la pasul anterior pot fi construite instanţe de obiecte ce vor putea fi accesate de la distanţă.

Pentru a putea face accesibil de la distanţă un obiect prin intermediul mecanismului RMI trebuiesc realizaţi următorii paşi suplimentari:

Instalarea în cadrul aplicaţiei ce urmează a construi şi înregistra obiecte distribuite a unui manager de securitate folosind clasa java.rmi.RMISecurityManager;

Lansarea în execuţie fie de la consolă, fie direct din aplicaţie a utilitarului rmiregisty. Lansarea de la consolă se face cu comanda rmiregistry port, unde port reprezintă portul pe care utilitarul va aştepta conexiuni. Lansarea din aplicaţie se face cu LocalRegisty.createRegistry (port).

Paşii pentru construcţia şi iniţializarea unui obiect distribuit sunt următorii:

- se construieşte obiectul pe baza clasei ce implementează interfaţa RemoteInterface;
- înainte de a instanţia şi a face vizibile în reţea obiectele distribuite, pe baza clasei definite la pasul anterior se generează clasele Stub şi Skeleton folosind utilitarul rmic.exe, specificând clasa care defineşte structura obiectelor distribuite. Aceste 2 clase implementează mecanismele de transmitere la distanţă a apelurilor metodelor şi a răspunsurilor lor.
- se înregistrează obiectul cu utilitarul rmiregistry pentru a putea fi accesat de la distanţă – Naming.rebid („//numeCalculator:port/numeObiect”) unde numeCalculator reprezintă numele calculatorului unde s-a activat rmiregistry, port portul pe care este lansat în execuţie rmiregistry şi numeObiect numele sub care obiectul este recunoscut şi invocat la distanţă.

Aplicaţia client

Pentru ca o aplicaţie client să obţină o referinţă la un obiect distribuit trebuie să se instaleze un gestionar de securitate de tip java.rmi.RMISecurityManager. După instalarea gestionarului de securitate clientul poate obţine o referinţă către un obiect distribuit folosind o instrucţiune de forma

```
ObjectInterface obj=Naming.lookup („//host:port/name”);
```

Unde host reprezintă numele calculatorului pe care obiectul distribuit este înregistrat, port reprezintă portul pe care utilitarul rmiregistry este lansat în execuţie pe maşina server şi name numele sub care obiectul a fost înregistrat în cadrul serverului.

După obţinerea referinţei către obiectul distribuit, aplicaţia client va putea manevra obiectul respectiv exact în acelaşi mod ca şi obiectele locale. Apelul metodelor din obiectul distribuit sunt de fapt transmise la distanţă prin intermediul mecanismelor RMI.

Mecanismele care se ocupă de apelarea la distanţă a metodelor distribuite şi întoarcerea rezultatelor metodelor apelate sunt implementate în cadrul claselor

Skeleton şi Stub generate de către compilatorul rmic.exe. Cele 2 clase trebuie să fie accesibile în cadrul aplicației client care încearcă să apeleze un obiect distribuit.

Exemplu. Aplicație distribuită

Se propune o aplicație care permite modificarea de la distanță a algoritmilor de control pentru un set de controlere, într-un sistem cu structura din *fig. 2.5*.

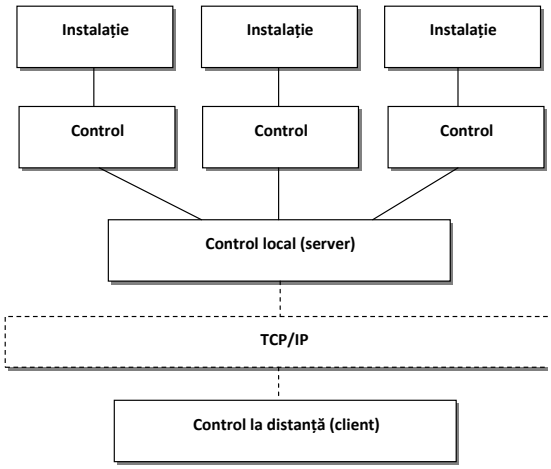

Figura 2.5 Arhitectura sistemului

Aplicația conține 2 componente: un controler local, care acționează ca un server şi un controler la distanță care acționează ca şi client. Componenta server permite inițializarea şi instalarea unuia sau mai multor controlere. Prin intermediul mecanismului RMI aplicația server permite instalarea în cadrul controlerelor de la distanță de către aplicația client a algoritmilor de control pentru fiecare din controlerele instalate.

Componenta client defineşte unul sau mai mulți algoritmi de control pe care îi poate instala în cadrul controlerelor aflate la distanță prin intermediul mecanismelor RMI. *Fig. 2.6* prezintă diagrama UML a claselor pentru aplicația server.

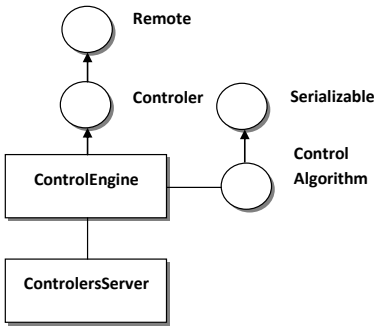

Figura 2.6 UML Server

În aplicația server clasa ControlersServer este responsabilă cu inițializarea mecanismelor RMI pentru a înregistra obiecte distribuite.

```
if (System.getSecurityManager==null)
{
        System.setSecurityManager (new RMISecurityManager
());
}
Registry reg=LocalRegistry.createRegistry (rmiport);
```

De asemenea, clasa ControlersServer defineşte metoda registerControler (ControlerEngine ctr) prin intermediul căreia un controler este înregistrat în sistem (folosind metoda Naming.rebind (...)) şi poate fi accesat de la distanță, precum şi metoda unregisterControler (String ctrlName) prin intermediul căreia un controler poate fi şters din sistem (folosind metoda Naming.unbind (...)).

Clasa ControlEngine defineşte structura obiectelor care vor putea fi apelate de la distanță, ea implementând interfața Controler în cadrul căreia sunt declarate

metodele accesibile de la distanță. Această clasă este de tip fir de execuție şi permite execuția în cadrul firului al unui algoritm de control. Setarea algoritmului de control se face prin intermediul metodei setAlgorithm (ControlAlgorithm alg) care poate fi apelată la distanță de către clienți distribuiți.

Clientul are posibilitatea de a defini proprii algoritmi de control şi de a-i instala în controler. El trebuie să apeleze la distanță metoda setAlgorithm (ControlAlgorithm alg) transmițând ca parametru metodei un obiect ce implementează interfața ControlAlgorithm.

Structura metodei run () ce se execută în firele de tip ControlerEngine este:

```
public void run ()
{
	nextState=RUNNING;
	while (active)
	{
		synchronized (look)
		{
			algorithm.executeStep ();
			if (nextState==PAUSED)
			{
				try
				{
					look.notify ();
					look.wait ();
					nextState=RUNNING;
				}
				catch (InterruptedException e1)
				{
					e1.printStackTrace ();
				}
			}
			try
			{
				Thread.sleep (1000);
			}
			catch (Exception e)
			{
				e1.printStackTrace ();
			}
		}
	}

}
```

Metoda setAlgorithm (ControlAlgorithm alg) apelabilă la distanță de către clienții distribuiți prin intermediul mecanismului RMI este:

```
public void setAlgorithm (ControlAlgorithm alg)
{
	if (controlerThread==null)
	{
		controlerThread=new Thread (this);
		controlerThread.start ();
	}
```

```
        else
        {
                nextState=PAUSED;
                synchronized (look)
                {
                        try
                        {
                                look.wait ();
                        }
                        catch (InterruptedException e)
                        {
                                e.printStackTrace ();
                        }
                }
        }
        synchronized (look)
        {
                this.algorithm=alg;
                nextState=RUNNING;
                look.notify ();
        }
}
```

Figura 2.7 prezintă diagrama UML a claselor client. Aplicația client implementează unul sau mai mulți algoritmi de control pe care îi poate încărca în cadrul controlerelor care rulează în alte locații. Comunicația între client şi controler se face prin intermediul mecanismului RMI, aplicația client putând obține o referință a unui controler pentru care apoi să seteze un alt algoritm de control prin apelarea metodei setAlgorithm (ControlAlgorithm).

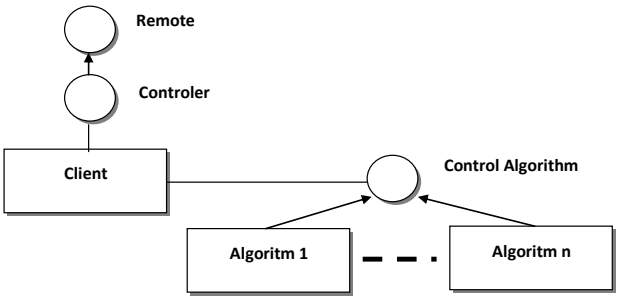

Figura 2.7 UML Client

Secvența de instrucțiuni necesară pentru ca un client să seteze un algoritm pe un controler distribuit este:

```
if (System.getSecurityManager ()==null)
{
        System.setSecurityManager (new RMISecurityManager);
}
Controler ctrl= (Controler)Naming.lookup (remoteObjectName);
Ctrl.setAlgorithm (new SimpleControlAlgorithm (args[1]));
```

[T. Leția, M. Hulea. Sisteme de control distribuit. Ed. Mediamira, 2005]